O'REILLY®

Broadview®
www.broadview.com.cn

流式架构

Kafka与MapR Streams数据流处理

Streaming Architecture

New Designs Using Apache Kafka and MapR Streams

[美] Ted Dunning Ellen Friedman 著

唐李洋 译

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O'REILLY®

流式架构

Kafka与MapR Streams数据流处理

Streaming Architecture

New Designs Using Apache Kafka and MapR Streams

【美】Ted Dunning　Ellen Friedman 著

唐李洋 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

所有连续的事件流都可以称为数据流。对连续数据流设计和构建流式数据架构，能够实现实时或近实时应用，提升整个组织的效率。本书以Apache Kafka 和MapR Streams为例，重点讲解如何确定使用流数据的时机、如何为多用户系统设计流式架构、为什么要求消息传递层具备某些特定功能，以及为什么需要微服务，并且描述了目前最符合流式设计需求的消息传递和流分析工具，适合架构师、大数据科学家及IT工程师阅读。

版权贸易合同登记号 图字：01-2016-8046

图书在版编目（CIP）数据

流式架构：Kafka与MapR Streams数据流处理 /（美）泰德·敦宁（Ted Dunning），（美）艾伦·弗里德曼（Ellen Friedman）著；唐李洋译. —北京：电子工业出版社，2017.7

书名原文：Streaming Architecture: New Designs Using Apache Kafka and MapR Streams

ISBN 978-7-121-31722-4

Ⅰ. ①流… Ⅱ. ①泰… ②艾… ③唐… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2017)第121174号

策划编辑：张春雨
责任编辑：徐津平
封面设计：Randy Comer　张　健
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：720×1000　1/16　印张：9　字数：100.8千字
版　　次：2017年7月第1版
印　　次：2017年7月第1次印刷
定　　价：55.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

前言

使用和处理连续数据流的能力，是一项极具竞争力的优势。因此，能够利用流数据，逐渐成为构建数据驱动型组织的一个重要条件。

流数据的广泛使用引发了如何进行更好的系统设计才能有效处理流数据的思考，涉及从多个数据源提取数据，以及各种不同的使用场景，包括流分析和持久化问题。

流架构设计的最佳实践层出不穷，甚至会让我们目瞪口呆——流系统设计的范畴已经远远超出服务于特定的实时或近实时应用。使用新的方法进行流设计，能够极大地提升整个组织的效率。

目标读者

如果你已经在使用流数据，并且希望设计出一种能够实现最佳性能的体系结构，或者正要探索流数据的价值，那么这本书应该对你很有帮助。本书提供了很多真实案例，帮助你理解如何将这些

方法应用到不同场景。此外，本书还为开发人员提供了示例程序的链接。

本书适合非技术或技术出身的读者，包括商业分析师、架构师、团队领导、数据科学家及开发人员。

内容梗概

本书内容包括：

- 如何确定使用流数据的时机
- 在多用户系统中如何更好地设计流架构
- 为什么这种设计要求消息传递层具备某些特定的功能
- 为什么流式架构支持微服务
- 最符合流设计需求的消息传递和流分析工具的描述

第 1~3 章阐述了流和微服务架构的基本知识。如果你已经对流数据的业务目标很熟悉，可以直接从第 2 章开始读，第 2 章描述了我们推荐的适合流系统的架构。

我们不仅解释了流架构最佳实践所需的能力，还介绍了一些目前能够满足这些要求的技术。第 4 章详细讲述 Apache Kafka，并提供了示例代码链接。第 5 章介绍另一种更适合消息传递的技术，即 MapR Streams，它使用 Apache Kafka API，但提供的功能更多。

后面的章节深入介绍了利用流数据的真实案例，并对这一激动人心的领域做出了前景展望。

排版约定

本书使用以下排版约定：

斜体（*Italic*）

表示新术语、URL、邮件地址、文件名及文件扩展名。

等宽字体（`Constant width`）

用于代码部分，以及段落内部的代码元素，如变量名或函数名、数据类型、环境变量、语句和关键字。

这个图标表示一般注解。

这个图标表示提示或建议。

这个图标表示警告或注意。

相关补充资料（示例代码、练习等）在这里下载：*https://www.mapr.com/blog/getting-started-sample-programs-apache-kafka-09* 以及 *https://www.mapr.com/blog/getting-started-sample-programs-mapr-streams*。

本书的目的是帮助你完成工作。一般来说，如果书中有示例代码，你可以在自己的程序和文档中使用这些示例代码。只要不是大批量复制这些代码，都不必联系我们请求许可。例如，借用书中若干块代码编写程序，不需要许可；而将 O'Reilly 书中的例子制作成 CD 售卖或发行，则需要许可。引用书中的示例代码回答某个问题，不需要许可；而在产品文档中大量使用示例代码，则需要许可。

我们非常希望你能在引用本书内容时标明出处，但并不强求。出处一般包含有书名、作者、出版商和 ISBN。例如："*Streaming Architecture: New Designs Using Apache Kafka and MapR Streams* by Ted Dunning and Ellen Friedman (O'Reilly).Copyright 2016 Ted Dunning and Ellen Friedman, 978-1-491-95392-1"。

如果你觉得示例代码的使用可能超越了合理使用范围，或者需要获得许可，请随时联系我们：permissions@oreilly.com。

Safari Books Online

Safari Books Online 是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品

技术专家、软件开发者、Web 设计师、商务人士和创意精英都可以将 Safari 在线图书作为他们的调研、解决问题、学习和认证的主要资料来源。

Safari Books Online 对于组织团体、政府机构和个人提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sam、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGarw-Hill、Jones & Bartlett、Course Technology 及其他数十家出版社的上千种图书、培训视频和正式出版前的书稿。要了解更多关于 Safari Books Online 的信息，请访问我们的网站。

联系方式

请将对本书的评价和发现的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

我们在 *http://bit.ly/streaming-architecture* 上列出了勘误表、示例和所有额外的信息。

要评论或者询问关于本书的任何技术问题，请发邮件到 *bookquestions@oreilly.com*。

要了解O'Reilly更多的图书、课程、会议和新闻，请访问我们的网站 *http://www.oreilly.com*。

我们的Facebook账号：*http://facebook.com/oreilly*

我们的Twitter账号：*http://twitter.com/oreillymedia*

我们的YouTube网址：*http://www.youtube.com/oreillymedia*

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务：

- **提交勘误**：您对书中内容的修改意见可在 *提交勘误* 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 *读者评论* 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31722

目录

第1章

为什么使用流

现实生活并不是批量化的。

很多系统监控和理解的对象是连续不断的事件流——心跳、洋流、机器度量值、GPS信号等。从根本上说，这样的例子无穷无尽。因而以数据流的形式从这些事件中收集和分析信息也是顺理成章的。即使是分析间断性事件，比如网站流量，也能得益于流数据方法。

将数据作为流（stream）来处理有很多潜在的优势，不过到目前为止要用好这种方法仍然有些困难。流数据（streaming data）和实时分析（real-time analytics）是相当专业的工程方法，而不是通用方法。那么，为什么现在大家对流的兴趣呈爆炸式增长呢？

简单的答案是，现在出现了很多新兴技术，能够以高性能的方式处理大规模的流数据，因此越来越多的组织开始以流的方式处理数据。大规模下的极高性能是最主要的技术进步之一，但不是全

部。以前，持久性消息队列的消息吞吐率大约是每秒数千个消息。而现在的新技术下，即使在消息持久化的时候，也能达到每秒数百万个消息的吞吐率。通过横向扩展（horizontally scale），系统还能实现更高的吞吐率。现代流系统带来的好处不止是规模扩展和性能提升。

从流数据获得实时的洞察力已经从理念变成实践。事实证明，流式架构（stream-based architectures）还有很多根本性的强大优势。

流数据不止适用于特别专业的项目。流式计算已经成为数据驱动型组织的规范。

新的技术和架构设计能够构建弹性系统，不仅更加简单有效，而且能够更好地对业务流程进行建模。可能是因为新系统解耦了传送数据和使用数据之间的依赖性。来自很多源（source）的数据，可以以流的方式集成到一个新型数据平台，供不同的消费者（consumer）即时使用，或者以后需要的时候再使用。诸多可能性耐人寻味。

我们将解释为什么这种广义视角的流架构（streaming architecture）很有价值，不过首先我们来看一下现在或不久的将来人们如何使用流数据。连续性数据最重要的来源之一就是物联网（Internet of Things, IoT）传感器，而物联网中发展最快的领域是车联网（Connected Vehicle）。

飞机、火车和汽车：车联网和物联网

现在和不久的将来，个人汽车很可能需要与若干不同的对象交换信息。这些对象可以是驾驶员、汽车制造商、信息提供商，也可以是保险公司、汽车本身，未来很快可以是路上行驶的其他汽车。

联网汽车（connected car）是车联网领域中发展最为迅速的方向之一，但这个概念并不是全新的。早在 20 世纪 70 年代早期，最早的联网汽车就引起了公众的注意，即NASA的月球考察车（Lunar Roving Vehicle, LRV），图 1-1 是它在月球上的照片。

以前，地球上的驾驶员使用纸质地图导航（假设他们驾驶时能够成功地折叠和打开这些地图），手动检查油、冷却剂和胎压；而 LRV 太空驾驶员的导航方式是，连续不断地将方向和距离数据发送给某台电脑，然后由电脑计算出任务所需的全部重要信息，包括整体方向以及回到登月舱的距离。这种“联网汽车”通过音频或视频传送的方式与地球通信。地面指挥中心的操作员能够从地球上某个位置启动和定向 LRV 上的视频摄像头，二者相距大概 25 万英里[注1]。

注 1　约 40 万公里。——译者注

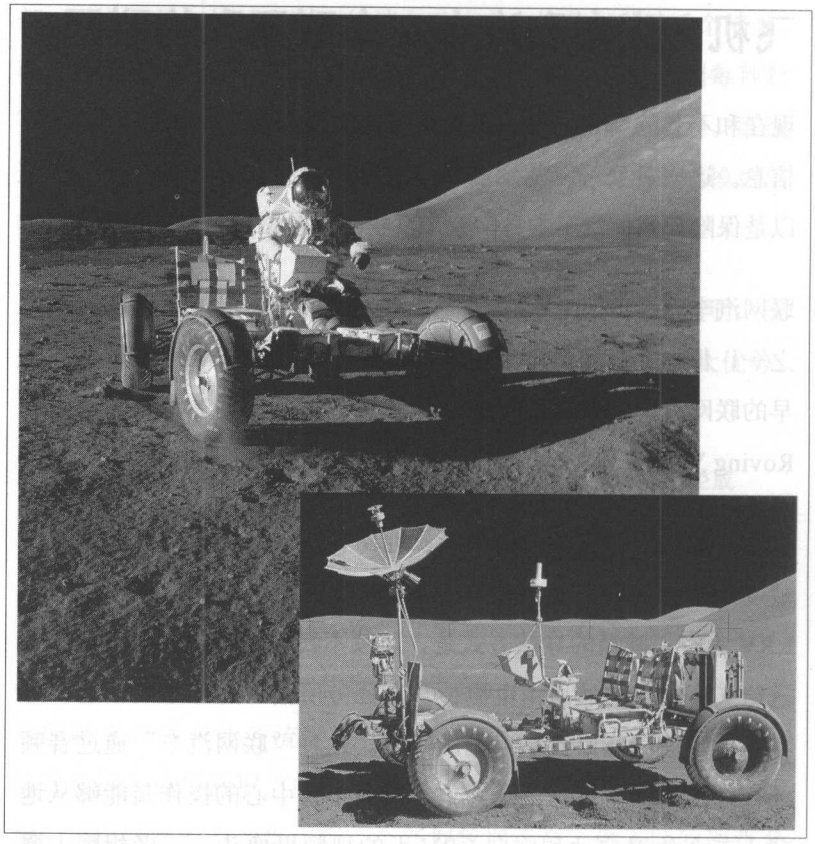


图1-1　上图：1972年阿波罗17号任务中，美国NASA宇航员和任务指挥官Eugene A. Cernan在LRV上考察月球的表面。在装载执行任务之前，该车的发动机被拆卸下来了（图片来源：NASA/宇航员Harrison H. Schimit；公共域名链接：*http://bit.ly/lrv-apollo17*）。下图：1971年阿波罗15号任务中配置齐全的LRV。这是一辆联网车，低增益天线负责音频数据，高增益天线负责视频数据，它们将这些数据传回地面指挥中心（图片来源：NASA/Dave Scott; 用户Bubba73剪裁；公共域名链接：*http://bit.ly/lvr-apollo15*）。

自从阿波罗任务以来，地面车辆的车联网取得了很大进展。令人

惊讶的是，机动车驾驶员呼声最高的联网服务是，收听自己的音乐播放列表，或者在开车的时候更方便地使用手机——好像他们想要一部车轮上的手机。联网汽车令人满意的服务还包括，从汽车生产商那里获取软件更新，比如正确发出操作不当警告信号的软件更新。较新型号的汽车能够在牵引和行驶的过程中利用周围环境数据做实时修正。汽车的功能性数据可用于预测性维护（predictive maintenance），或者向保险公司发送关于驾驶员和车辆行为的警告。（在本书编写时，虽然新型联网汽车已经在使用4G网络，但它们还不能与月球通信。）

如今，汽车还配有行车记录仪（Event Data Recorder，EDR），又称“黑匣子”，飞机上都有这样的设备。多种多样的参数产生了大量的传感器数据，这些数据被采集和存储，主要用于事故或故障的分析。

对于高性能汽车来说，联网尤为重要。一级方程式赛车就是联网汽车。新型一级方程式赛车有上百个传感器，测量频率高达1kHz（采用最新技术的话更高），通过RF链路将数据发送到检修加油站，然后发给总部。

车联网不仅涉及汽车，火车、飞机和轮船也能利用传感器数据、GPS跟踪等。例如，英国铁路、思科系统和电信公司之间正在合作构建互联系统，以降低英国火车出事的风险。通过装配大量传感器，火车能够监控轨道，而轨道在与操作中心通信的同时也能监控火车，以连续的数据流传送火车速度、位置、运行及轨道情况等数据信息，使得在事件发生时以低延迟的方式提供洞察力成

为了可能，这样，工程师们就能及时采取行动。

这些例子都强调了流数据实时分析的主要优势：对事件作出快速响应的能力。

流数据：这才是现实世界

妥善处理流数据不仅能够获得即时的、可操作的洞见，同时也是最普遍追求的目标之一。很多情况下，为了突显响应的价值，需要快速响应。例如，一个叫 Waze 的手机应用，以众包的方式提供导航并更新交通状况。如图 1-2 所示。Waze 从数百万个驾驶员那里获取实时的流输入，给出当前交通状况和道路信息的报告。这些实时的洞见有助于驾驶员确定行驶路线，从而减少油耗、驾驶时间和烦躁情绪。

假如早班高峰期某段高速公路发生事故造成拥堵，如果能在发生事故和拥堵的当时通知驾驶员，就非常有价值。反之，如果在事故发生一小时后，或者当天晚上才告诉驾驶员，那就没什么价值了，除非你想查看历史交通情况。这种事后洞见对于避免早班高峰期拥堵并没有什么帮助。Waze 就是一个体现信息的时间价值（time-value of information）的简单例子：某些特定知识的价值随着时间的流逝迅速降低。这个导航工具的关键在于，通过 4G 网络处理流数据，然后及时发送给驾驶员。

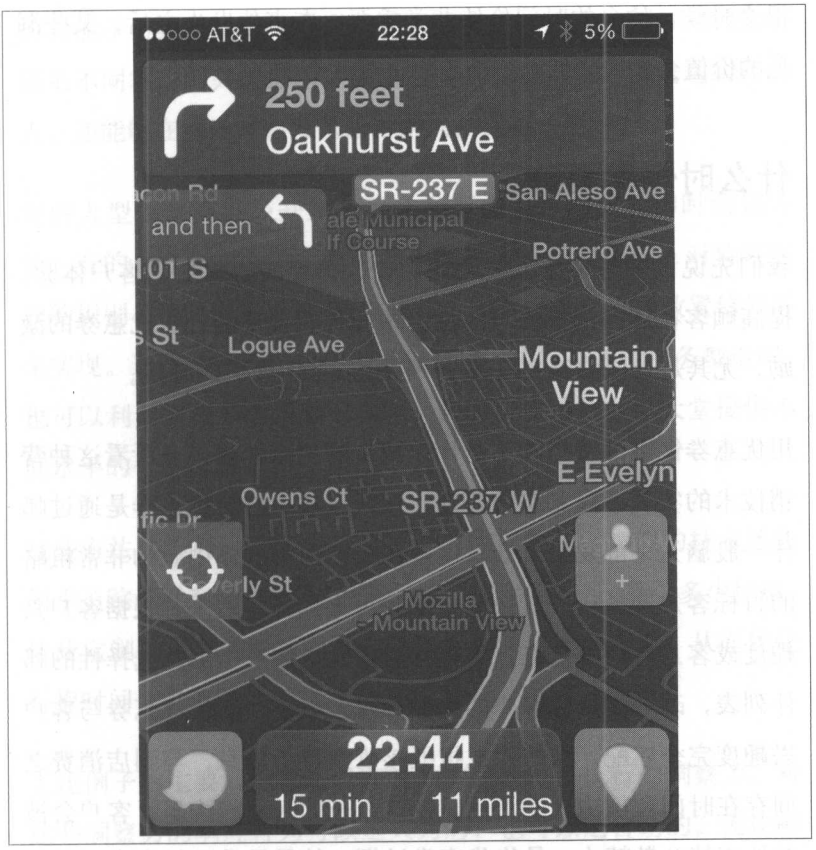


图1-2 智能手机应用Waze截屏。不仅提供点对点导航，还能通过数百万个驾驶员之间的实时交通信息共享实现更多价值。

流数据的低延迟分析允许我们在事件发生时作出响应。

很多时候，信息的时间价值非常重要，在事件发生之后，某些洞见的价值会迅速降低。下面我们再看几个例子。

什么时候需要流

我们先说说零售市场。假设我们要改善瓷砖水泥店的客户体验，提高顾客在店内消费的可能性。顾客可能会受到打折优惠券的激励，尤其是当他们确实被某个产品或服务所吸引的时候。

用优惠券促进消费当然不是什么新奇的主意，我们来看看这种营销技术的实现手段和有效程度。很久以前，打折优惠券是通过邮件一股脑儿全部发给公众，这是在大范围的群体中做的非常粗略的目标客户选择——就像消防水管灭火那样。后来，根据客户兴趣度或客户行为等信息，优惠券会被发送到一个更具选择性的邮件列表，改进了目标客户的选择方式。但是，即使优惠券与客户兴趣度完全匹配，从邮件或报纸接收到打折信息，与到店消费之间存在时间和精力上的滞后。在这么长的时间鸿沟里，客户会被其他事情分散精力，最终优惠券过期，结果导致虽然命中了目标客户却还是错过了销售机会。

以前逛商店的时候，客户会被特定款式的毛衣或包包的打折广告牌所吸引。而现在，当客户走到电子产品区域的时候，手机上就会自动跳出折扣优惠码。再比如，某家户外服装店基于客户的历史购买记录或网页浏览习惯，判断哪些客户喜欢野营和划船，哪些客户喜欢野营和山地骑行。当客户进店的时候，信号塔与客户

的智能手机通信，以短信的方式将折扣信息发送到客户手机上，满足不同客户的兴趣需求。如果这些折扣券不仅能够发送给对的人，还能够在对的时刻发送给对的人，岂不是更好？

有些大型零售商已经开始实现这种面向客户响应的即时营销方法，有的是自主研发的，有的是由第三方提供服务的。对特定客户的识别，可以利用手机 WiFi 连接，或者通过在店内放置信号塔来实现。这些技术不仅适用于零售商店、旅馆及其他服务型组织，也可以利用这些方法识别回头客，或者在旅馆前台或大堂提供不同水平的个性化服务。

这些方法并不局限于零售行业。令人惊讶的是，类似的技术还能用于追踪垃圾车的位置。“智能”垃圾桶能知道自己装了多少垃圾，依此定制垃圾车的路线规划，以更好地满足实际需求，从而优化行程时间、降低油耗并提交设备使用率。

上述例子的主要目标都是以即时的方式获取可执行的洞察力。对这些洞察力的响应行为可以是人为的，也可以是自动的。无论哪种方式，时间都是关键，目的是利用流数据和新技术，立即做出响应。其实，流数据带来的好处还不止这些，稍后我们再讨论。现在，流架构已经成为很多过程的核心，其中的一些过程甚至之前从未考虑过能以流方式实现。

最重要和常见的一种情况是，对流数据进行低延迟分析（low-latency analytics），这是保护数据安全的一种重要能力。对于一个

设计完善的项目来说，系统中发生的各种事情都是可监控的。例如信用卡交易或者登录银行网站相关的事件序列等。采用异常检测（anomaly detection）技术和超低延迟技术，能够迅速发现人为的或机器人网络攻击，从而采取行动以阻止入侵或减少损失。

批 vs. 流

过去，为了进行大规模数据分析，需要以批量的形式收集和分析数据。那么，批过程和流过程的区别是什么？我们做一个简单的比喻：将数据比作水，批过程和流过程分别相当于用桶装水然后交付给用户，以及用水管让水流向用户。

可以在水管上加个阀门，关闭水龙头的时候水流被周期性截断。有了水管和阀门，用户就可以选择截断水还是让水继续流——能够同时应对两种交付方式。反之，即使用桶运水的速度足够快，这种桶（批）式交付也永远不可能是连续的流。

从计算上讲，批处理非常适合处理海量规模的分布式数据，批式计算方法，如 MapReduce 或 Spark，在很多情况下仍然非常有用。如果对一系列事件按小时汇总，或者按日或按周求和，那么批处理能够满足要求。然而很多时候，批处理并不足以反映真实情况。因此，流式计算越来越受关注，这将在第 3 章详细解释。

前面提到，采用流的方式处理数据，其好处不止是实时或近实时分析从而即刻获取洞见。有时候还要求持久性（durability）：需要一个存储事件流数据的消息传递系统（message-passing system），通过检查点（checkpoint）实现从流的特定位置重新读取。

不止是实时：流架构的更多优势

工业界有很多物联网（IoT）案例，其中流数据的价值多种多样。现在，泵和涡轮机等设备装有传感器，实时或近实时地提供连续的事件数据流以及参数测量值，很多新的技术和服务也正在研发，采集、传输、分析和存储这些物联网数据。

现代制造业正在革新，更加强调灵活性、数据驱动决策的快速响应能力，以及产品或过程变更的重新配置。未来，设计、工程和产品团队的协作将会更加紧密。英国谢菲尔德大学－波音公司先进制造技术研究中心（AMRC）的先进技术印证了这一点。一个完全可重构的未来工厂“Factory 2050”在 2016 年正式启动。其设计理念是，将来自不同公司的产品“接”到工厂的环形结构上，进一步实现定制化。这个设备如图 1-3 所示。

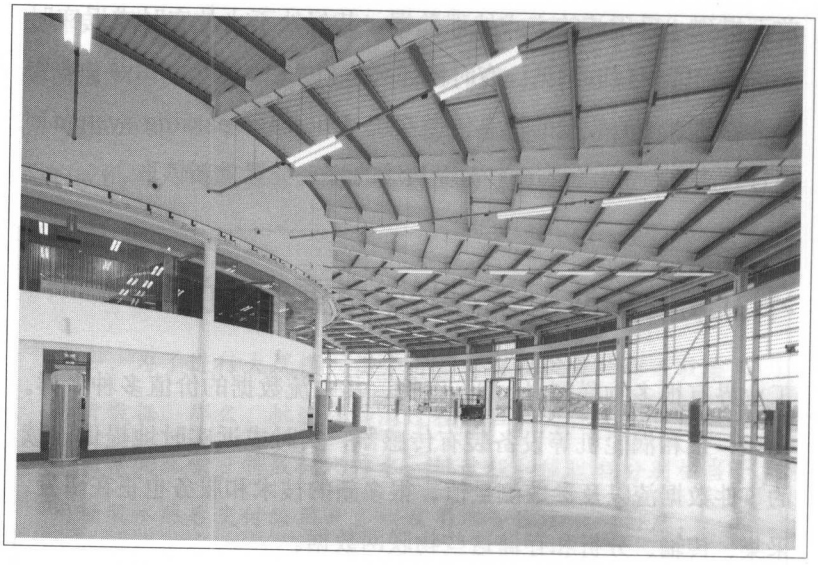


图1-3　完全可重构的Factory 2050的中心辐射型（hub-and-spoke）设计，这个革命性的设备来自AMRC。这个灵活的室内装备支持产品设计的快速更新，是一种全新的制造风格（感谢架构师Bond Bryan提供图片）。

当下流行的“智能零部件”（smart part）产品也反映了物联网制造业的灵活性。其理念是，不仅生产过程中的传感器测量值能够提供一个关于生产过程的细粒度视图，而且生产出来的零部件也能在应用部署之后向生产商反馈信息，告诉生产商零部件在整个生命周期的性能情况，这反过来又会影响设计和生产。此外，这些智能零部件的报告流本身也是可盈利的产品。生产商可以出售这种从数据获得洞察力的服务，或者出售数据本身，再或者许可数据访问权限。所有这些无不表明，流数据对于物联网的成功来说必不可少。

传感器数据流的价值不止是实时的洞察力。综合考虑泵或其他工业设备的零部件传感器数据，以及长期、详细的维修记录。传感器数据的事件流就像是“时间机器”，通过机器学习模型回顾历史，在故障之前从测量值中发现异常模式。再结合零部件的维修历史信息，就能在事件之前发现潜在故障，从而在灾难性故障发生之前做出预测性维修警告。这种方法不仅省钱，有时候还能救命。

流架构的最佳实践

关于流数据的旧式思维是“用完即作废”。假设我们有一个实时分析应用，通过处理流信息，实时更新仪表盘，然后就丢弃这些数据。其中，上流消息队列系统就好比是一个临时保存事件数据的安全缓存，防止数据流分析应用的短时间中断。其理念是，事件流数据只对实时分析有价值，或者说，不容易或不值得保存这些数据，因为它们是不断变化的。

队列保障了消息的安全性，而且如果采用适当的消息传递技术，队列还能够实现更多。要想得到流数据的全部好处，我们需要摒弃这种“用完即作废”的思考方式。

当我们谈到流数据的时候，不要认为应该用完就丢掉它。数据流的持久化（persistence）是有实在好处的。

实时响应现实世界是一种强大的能力，而流系统使其成为可能。为了更好地发挥流系统的更多优势，除了实时分析的计算框架和算法以外，我们有必要了解更多信息。近年来，低延迟、内存式框架已经取得了很多激动人心的进展。这些流处理分析技术极其重要，第 2 章会介绍几个很棒的新工具。不过，要想有效地使用这些技术，还需要访问合适的数据——以流的方式收集和传输数据。以前这种做法并不普遍。但是现在情况变了，而且变化得很快。

如今，现代系统能够更加容易地处理流数据流，原因之一是消息传递系统得到了改进。高效的消息传递技术从很多个源——上百、上千甚至上百万个源——收集流数据，然后交付给多个数据消费者，比如实时应用及其他数据消费者。有效的消息传递能力应当是流基础设施的一个基本要求。

现代流架构设计的核心是消息传递能力，它使用多个源的流数据、为多个消费者提供服务。有效的消息传递技术能够解耦数据源和数据消费者，这是敏捷的关键。

医疗数据流案例

医疗行业是多个消费者在不同时间使用相同数据流的完美案例。图 1-4 展示了检测结果数据流的几种可能用途。在这个例子中，

医学检测有多种数据来源，如心电图、血板或磁共振成像仪等。我们采用现代消息传递技术来处理医学检测流，即图中的水平圆管。医学检测结果数据流不仅包括测试结果，还包括病人编号、测试编号，可能还有测试设备的编号。

有了流数据，首先想到的可能就是实时分析，图 1-4 中以“A”表示实时应用这种消费者。在流数据的传统用法中，数据的用途是单一的：实时应用读完数据之后随即丢弃。但是，在新型流架构设计中，除了实时分析程序以外，可能还有多个消费者马上需要这些数据。例如，“B”类消费者需要电子病历数据库以及每台设备执行的检测次数（用于设备管理）。

有趣的是，我们希望这个数据流能够作为一种持久性、可审计的检测结果保存下来，从而服务于多种用途，比如保险审计（即图 1-4 中的“C”）。由于后期才需要审计，所以当时可能还没有规划它。如果消息传递软件支持数据记录的持久性（durable）和可回放性（replayable），那实现审计就不是问题。

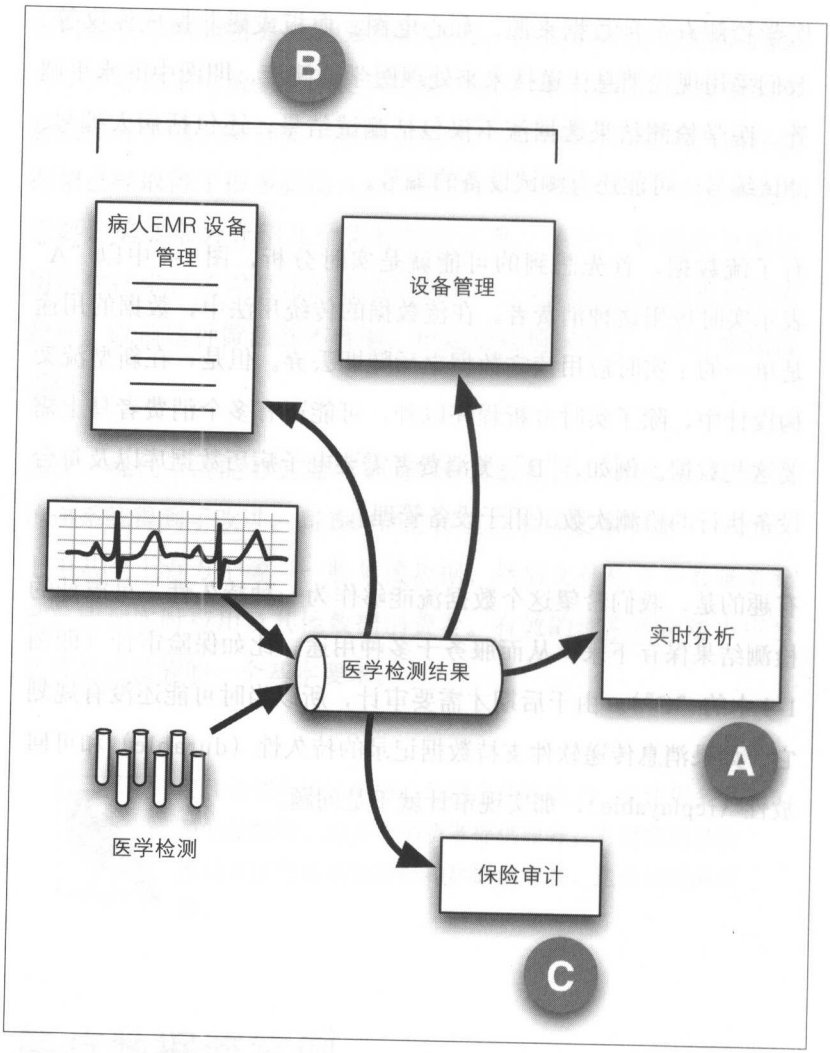


图1-4　医疗流数据不仅能用于实时分析。这是系统的原理设计图，来自多个源的数据以不同的方式、在不同的时间，为多个消费者所用。这里用圆管表示消息传递技术，文字标记表示数据流的内容（即医学检测结果数据流），EMR表示电子医疗记录。注意看C类消费者“保险审计”，可能在系统设计或部署的时候并没有规划这个应用。

流数据：架构设计的核心

本书探讨流数据（streaming data）的价值，解释为什么以及怎样正确地使用流数据，并提出了流架构（streaming architecture）设计的最佳实践。构建有效的流数据系统，需要记住以下关键点：

1. 对流数据进行实时分析，在事件发生时及时作出响应并提供洞察力。

2. 不要丢弃流数据，数据持久化的回报是多种多样的。

3. 通过适当的技术，可以将流数据复制到地理上分散的数据中心。

4. 有效的消息传递系统远远不止是一个服务于实时应用的队列：消息传递系统是整个大数据架构设计的核心。

后面三点（流数据的持久化、地理分布式复制以及消息传递层的核心地位）是相对较新的流架构设计理念。这里最具颠覆性的概念是，流架构不应该局限于实时应用。如果我们将这种流方法作为整体架构的核心，组织将受益良多。

第2章

流式架构

上一章我们讨论了为什么人们对流数据越来越感兴趣。现在我们解释如何构建最有效的流系统。

消息传递技术的发展让我们几乎随处都可以使用流。这也是本章要传达的最重要的思想。

在企业数据活动中部署流式架构将带来巨大的好处。

新的架构设计需要大规模地转变构建系统的整体设计方法。这种转变不仅是指要掌握特定技术或技能，还包括更广泛、更基础层面的转变。而且，这种系统架构的进化是不同寻常的，它以增量的方式被引入，随着越来越多的服务实现转变，收益将成倍增加。

狭义视角：实时应用

整体向流架构转化的需求可能并不迫切和明显。使用流数据的最初动机或许是某个项目或业务目标需要进行实时分析。例如，假设要构建一个实时更新的仪表盘。首先确定相关的流数据源，需要使用一个强大的流处理软件，比如 Apache Spark Streaming。我们需要它的内存功能提供近实时处理，从而满足特定需求。然后我们会将这个分析应用的结果导出到仪表盘，实现近实时更新。而且，我们还喜欢这样的设计理念：不需要将流数据保存到文件或数据库，就能够立刻分析原始流数据。你可能以为这样就已经成功了。

但是，假设分析程序出现了临时中断，或者运行速度变慢，进来的数据流可能已经被丢弃了。为了保险起见，我们在数据到 Spark 应用之间安排一个消息队列作为安全缓存。图 2-1 展示了这种仅用于实时流处理的单一目标数据路径设计。在本章中，我们将关注不同案例中设计模式的范型。

如果选择合适的工具实现队列和分析工作，那么按照图 2-1 构建仪表盘没什么不好，而且单就实现这一个目标来说，这已经相当不错了。但是，如果要全面利用数据改善整体的管理、运营和开发活动，则需要更好的系统设计。

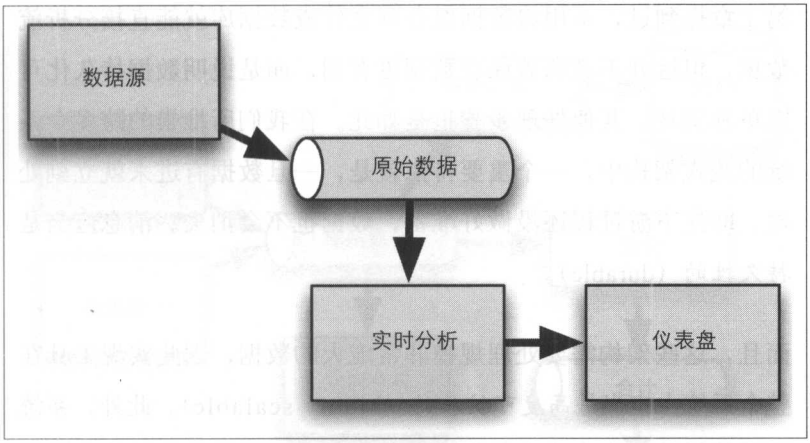


图2-1　以前实现实时分析的典型设计图。来自单一源的数据，用于更新实时仪表盘。圆管表示消息传递系统，用于保障数据的安全性。

我们建议彻底颠覆系统设计的思路。想法是，*将数据流贯穿到整体架构中去*——数据流是处理数据的默认方式，而不是特殊方式。目的是将整个操作过程设计成流线型（不是双关语），在需要的时候更容易地将数据提供给消费者使用，实现实时分析及更多应用，而且没有大量管理上的负担。

通用流式架构的关键问题

相比于传统的数据处理方式，从数据中获得实时的洞察力然后持久化是一个巨大的转变。而且，人们正在研发基于流算法的机器学习模型，实时地做出数据决策的同时还能学习。对这些系统来说，“快”很重要，因此内存式处理的方法和技术获得了越来越多的关注。

第 1 章提到过，不用将数据保存到文件或数据库就能直接分析流数据，但这并不意味着保存数据没有用，而是说明数据持久化可以单独实现。其他处理步骤也是如此。在我们所推崇的跨多个系统的流式架构中，一个重要特点就是，一旦数据有进来就立刻处理，即使下游过程还没做好准备，数据也不会消失。消息应当是持久性的（durable）。

而且，这些架构需要处理规模非常庞大的数据，因此实现工具在整个系统中应当是高度可扩展的（highly scalable）。此外，系统还要能够处理来自多个数据源的数据，为不同的数据消费者提供服务。

> “数据集成是指将组织拥有的全部数据提供给需要的全部服务和系统。”[注1]
>
> ——Jay Kreps

这种系统设计方法将流数据作为整体数据集成的一部分，其重要优势之一就是，能够对不断变化的需求迅速做出响应。解耦（decouple）数据源和数据消费者之间的依赖性，是实现这种灵活性的关键，第 3 章讨论流数据和微服务的时候再详细解释。

图 2-2 给出了流式架构的概念图，为了保持简单性，图中省略了一些细节。我们对图 2-1 单一目标设计的实时更新仪表盘进行了改进和扩展。为了方便对比，图 2-1 中的组件在图 2-2 中以阴影表示，而没有阴影的部分表示新添加的项目以及对原始数据流的修正。

注 1　*I Heart Logs*，O'Reilly

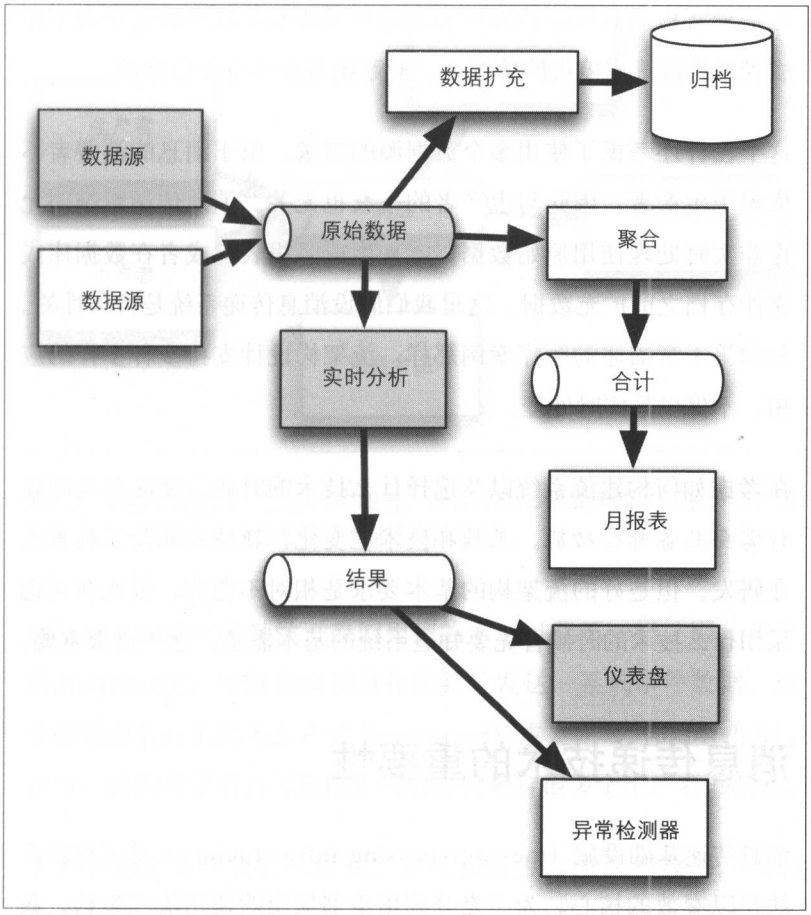


图2-2　流架构总体设计概念图：不止一个组件使用同一个消息流，它们的用途各不相同，不仅用于实时分析。这种设计支持数据集成，整个过程采用流消息传递基础设施按需交付数据。

注意，实时应用的结果首先流向一个消息流（图中用圆管表示），然后才到仪表盘，而不是直接到仪表盘。这样，这个结果就很容易被其他组件使用，比如本例中的异常检测模块。这种设计的好

处是异常检测器可以后来再加。这种灵活的系统设计允许在不造成管理性困扰或停机的情况下，对系统设计本身进行修改。

这个设计还考虑了使用多个数据源的需求。由于消息的消费者不依赖于生产者，因此与生产者的个数也无关。消息传递系统还允许非实时处理使用原始数据，比如生成月报表，或者在数据库或文件存档之前扩充数据。这里我们假设消息传递系统是持久性的。如同第 1 章描述的医疗案例那样，流架构设计支持多种多样的应用，不仅仅是实时处理。

在考虑如何构建流系统以及选择什么技术的时候，要注意这种设计需要具备哪些功能。工具和技术在变化，新技术和新工具也正在研发。但是好的流架构的基本要求是相对不变的，因此在考虑采用什么技术的时候首先要知道系统的基本需求，这一点很重要。

消息传递技术的重要性

消息传递基础设施（message-passing infrastructure）是这种新方法得以奏效的核心。为了充分利用本书提出的通用流式架构，我们考察一下消息传递组件的关键功能。图 2-3 展示了消息传递层的工作原理。

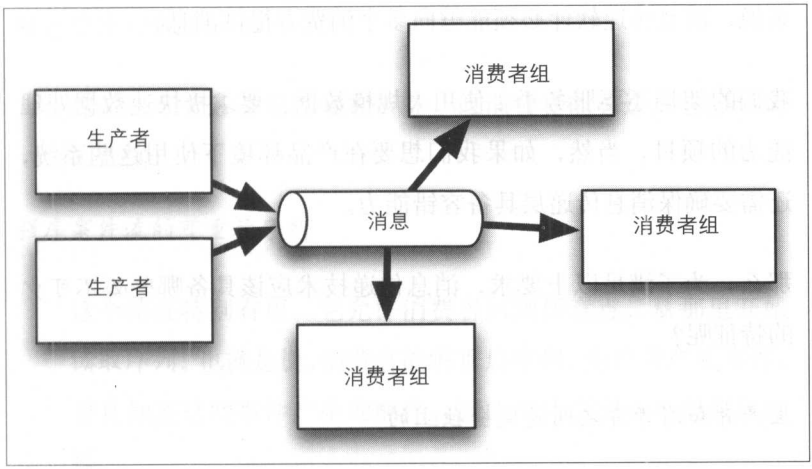


图2-3 消息传递技术，即图示的圆管，处理来自多个数据源（即生产者）的数据，并以解耦的方式由多个消费者组订阅使用。

在描述不同系统的时候，我们用到的术语可能不一样，因此要考虑其内在涵义。数据源向消息传递系统发送一系列事件数据。这个数据源有时也称为*生产者*（producer）或*发布者*（publisher）。这里，我们希望消息传递技术能够处理来自很多个生产者的消息。

生产者传递事件数据的时候，并不知道谁要用这些数据。我们称使用消息的那一方为*消费者*（consumer），有时也称*订阅者*（subscriber）。每个消息流都会被命名，称为*主题*（topic）。消费者（或消费者组）根据需要请求或订阅主题。第4、5章将进一步解释这些术语。这种方法的优势在于，不论消费者是否做好了接收的准备，都可以给他们发送消息，而且直到消费者准备好之前消息一直都是可用的。这是消息传递系统的本质特征。对于我们的设计

来说，消息传递软件必须能够向多个消费者提供消息。

我们的架构还要服务于：使用大规模数据、要求极快速数据处理能力的项目。当然，如果我们想要在产品环境下使用这些系统，还需要确保消息传递层具备容错能力。

那么，为了满足以上要求，消息传递技术应该具备哪些必不可少的特征呢？

生产者和消费者之间是完全独立的

> 消息传递工具一定不能要求生产者知道哪个消费者处理消息。

持久性

> 这是生产者和消费者完全独立这一条件的隐含意思。否则，如果生产者和消费者在数据进来时不能协调交付，消息就会消失。

每秒极高的消息处理速率

> 流数据的使用场景都要求极限性能。如果将消息传递作为系统的核心中枢，就不得不应对极高的消息速率。

通常来说，消息传递系统通过持久化来保持生产者和消费者之间完全隔离时，都会带来速度上的损失。然而，对于通用流式架构来说，以上特征必须同时存在。

对主题进行命名

这是一个普通却很重要的特征，使消费者能够选择它们所需的数据。

强序事件流的可重放序列

这个特征特别有用。它允许消费者回到任意点，从那里开始读取序列。也就是说，消费者能够重启序列。生产者产生事件，并且知道这些事件产生的顺序，因此事件之间存在逻辑依赖性。

容错性

对重要的关键系统来说，显然需要具备这个特征。

地理分布式复制

不是每一个场景都要求这个功能，但很多时候这是一种绝对需求，因为架构需要在不牺牲上述各个功能的前提下，在不同地理位置的多个数据中心之间实现系统功能。

去哪里找能满足这些苛刻要求的消息传递工具呢？有两个绝佳的工具能够满足通用流式架构的要求：(1) Apace Kafka，将在第 4 章详细介绍；(2) MapR Streams，它使用 Kafka API，将在第 5 章讨论。

从某种程度上说，消息传递工具可以分为两大类：Kafka 类（Kafka 和 MapR Streams），以及其他。

Kafka 的工作原理与 Apache ActiveMQ 或 RabbitMQ 这样的传统消息传递系统差别很大。最大的一个区别就是，对于传统系统来说，持久化是一个成本高昂的可选项，它通常会带来两个数量级的性能下降。相反，对于 Kafka 或 MapR Streams 系统来说，即使单个服务器每秒处理的消息量高达 GB 级甚至更多，系统仍然能够自动持久化所有消息。造成这种性能上巨大差异的原因之一是，Kafka 类系统不支持逐个消息确认（message-by-message acknowledgment），而是让服务按照顺序读取消息，偶尔根据上一次未读消息的偏移量更新游标。而且，Kafka 侧重于消息处理，而不是数据转换或任务调度。这种范围上的限制有助于 Kafka 实现非常高的性能。

实时分析工具

各种流数据处理技术的发展，以及具备高度扩展性的消息传递工具的演化，是越来越多的组织机构从流数据中发现实时洞察力的驱动力。到目前为止，书中使用术语“实时”（real time）表示相对低延迟，但是近似实时，与真正基于实时或超低延迟流进行数据分析，在技术实现上是不同的。对很多应用来说，依据服务水平协议（SLA），这种区别并不是特别重要。不过有些情况下确实

需要做到“实时”。

流分析（streaming analytics）技术和方法的具体细节已经超出了本书的范畴，这里我们只是大概介绍功能需求，以及几个工具和相互之间的区别。首先简单介绍 4 个相关技术：Apache Storm、Apache Spark Streaming、Apache Flink 及 Apache Apex。然后，我们看看流式架构下的分析最需要哪些关键功能。此外，我们还会对提供这些功能的技术做对比。

关于 Hadoop 的困惑

随着快速、内存式计算框架（如 Apache Spark）的发展，我们对 Apache Hadoop 和 Hadoop 生态系统产生了一些困惑。有人说 Hadoop 已经被 Spark 取代了，或者不明白为什么仍然需要 Hadoop。究其原因可能是，对很多应用来讲，像 Spark 这样的内存计算引擎实际上已经取代了 Hadoop MapReduce 这样的批处理计算框架。但是，Hadoop 的其他组件，例如 YARN 和 HDFS，仍然应用十分广泛。

这种困惑源于人们常常不太清楚 Hadoop MapReduce 和 Hadoop 相关技术生态系统的区别。Hadoop 生态系统项目包括 Apache Spark、Apache Storm、Apache Flink、ElasticSearch、Apache Solr、Apache Drill、Apache Mahout 等。这些项目是超大规模、经济高效的分布式系统的领头羊。

随着每个项目的不断演进，它们的功能也在不断变化。但是，能够为流分析提供最佳支撑的特征是相对不变的。

鉴于这些项目的功能都在持续演进，我们对于某种技术的表述和对比只代表特定时期的状态，但这些项目有助于我们理解流分析的具体功能特征。

Apache Storm

Apache Storm 是大规模分布式系统实时处理的先驱。该项目网站上说，“Storm 之于实时处理，相当于 Hadoop 之于批量处理。”Hadoop 的计算框架 MapReduce，在大规模批处理上受众广泛，而 Storm 是 Hadoop 生态系统中负责实时处理的部分。Storm 项目最初是由 Nathan Marz 在 Apache 之外创建的，后来慢慢成长为 Apache 顶级项目。

Storm 对无限的（unbounded）流进行实时处理。支持多种语言。近期，Storm 增加了窗口（window）功能，确保“至少一次”（at-least-once）消息交付，不过在处理纯转换的时候，或者在应用层而不是平台层定义窗口时，Storm 性能最佳。目前，Storm 的设计理念涉及“早期汇编”（early assembly），即以 Java 对象表示数据行，仅在读取数据的时候才真正创建对象。比起 Flink 这样使用字节代码引擎的系统，这种方式限制了性能，好像它们还在做别的什么事情一样。

分布式系统面临的挑战之一是，如果要在可能发生故障或者间断性操作的情况下完全保证操作的正确性，那么情况就复杂得多。至少一次（at-least-once）、至多一次（at-most-once）和正好一次（exactly-once）交付，可以部分实现这样的保证。至少一次处理方式是指每次处理一条记录，有时候一次也可以处理多条记录。至多一次处理基本上正好相反：一次最多处理一条记录，有时候甚至会丢失记录。严格地说，无条件地保证正好一次处理是不可能的，但是如果我们能够接受某些限制条件的话，可以做到看起来像是正好一次消息交付，这在实践中也够用了。例如，如果消费者只是使用消息简单地往数据库写入（但绝不是覆盖）一个值，那么一次接收多条消息与正好接收一条消息，其实没有什么区别。

Apache Spark Streaming

Spark Streaming 是 Apache Spark 下的一个子项目。Spark 起源于 2009 年 UC Berkeley AMPLab 的一个大学项目。2013 年，这个项目进驻 Apache 基金会，并于 2014 年成为顶级 Apache 项目。从 2014 到 2017 年以来，整个 Spark 项目已经得到了广泛关注和普遍应用。

Spark 能够将数据装载到内存然后反复查询，这种快速计算的创新促进了 Hadoop 生态系统计算方式的演进。Spark Core 使用一种称为弹性分布式数据集（Resilient Distributed Dataset, RDD）

的抽象结构。当作业太大以致内存放不下时，Spark 就将数据外溢（spill）到磁盘。Spark 需要一个分布式数据存储系统（如 Cassandra、HDFS 或 MapR-FS）以及一个管理框架。Spark 支持 Java、Python 和 Scala。

Spark Streaming 使用微批处理方式（microbatching）近似实现实时流分析。即批程序以很短的时间间隔处理近期的全部数据，并且保存以前数据的状态。尽管这种方法不是很适合低延迟（“真实时”）应用，但仍不失为一种将批处理扩展成近实时处理的明智之举，适用于很多应用场景。而且，批处理应用的代码同样完全适用于流应用。比起真正的实时系统，Spark Streaming 更容易保证正好一次（exactly-once）处理。如果需要更短延迟的分析（即实时分析），通常混合采用 Spark Streaming/Spark Core 和 Apache Storm 进行实时处理。

Apache Flink

Apache Flink 是一个高度可扩展的、高性能的处理引擎，能够处理低延迟及批量分析问题。Flink 是一个比较新的项目，起源于德国和瑞士的几所大学合作的名为 Stratosphere 的联合项目。2015 年，该项目进入 Apache 孵化器之后，更名为 Flink（在德语中意为“敏捷”或“迅速”）。如今，Flink 已经成为顶级 Apache 项目，拥有一支来自全世界的协作团队。随着关注的人越来越多，Flink 的知名度迅速增长，而且有些公司已经将其应用于产品中。

Flink 能够处理低延迟、实时分析应用，就像 Storm 擅长的那样；不仅如此，Flink 还能做批处理。实际上，Flink 将批（batch）视为流（streaming）的一种特例。Flink 程序是开发者友好的，以 Java 或 Scala 编写，保证正好一次（exactly-once）交付。与 Spark 或 Storm 一样，Flink 需要一个分布式存储系统。Flink 能够在高效处理海量数据的同时，确保“实时”水平的延迟。

Apache Apex

与 Apache Flink 一样，Apache Apex 也是一个可扩展、高性能的处理引擎，同时支持批量处理和低延迟流处理。Apex 最初诞生于企业界，前身是 DataTorrent RTS，后来内核引擎被开源，2015 年夏天该项目进入 Apache 项目基金会孵化。Apex 说自己是“用 YARN 的思路开发出来的”。因此，它运行起来其实是一个 YARN 应用程序，但与 YARN 的功能不重复。Apex 支持 Java 或 Scala 编程，旨在为 Java 程序员提供更为便捷的方式构建大规模数据应用，以及重用 Java 代码。与其他流分析工具一样，Apex 需要一个存储平台。Apex 的特殊优势在于其 Malhar 函数库涵盖了大量分析需求。

流分析功能比较

这里我们介绍的流分析工具并不全面，也不权威。我们说过，这些技术都在不断发展，每个项目及其特性和能力描述也不是一成不变的。因此，我们强调这种架构风格的功能性影响，如果有更

多选择我们建议继续进行评估。尽管如此，通过简单对比这些关键功能，还是很有帮助的，如下。

基本要求

这种架构风格中的分析技术必须是高度可扩展的（highly scalable），能够在不丢失信息的情况下开始和结束，并提供与消息传递技术（类似于前面的 Kafka 和 MapR Streams）之间的接口。

性能和低延迟

这两者是相对的。就最佳实践来说，现代体系结构通常需要应对批应用和超低延迟的流应用，以满足已有应用的需求，或者以备未来不时之需。而且，通常也要求高性能。

既能批处理又能提供低延迟，而且在不牺牲性能的情况下实现实时处理，这样的技术非常具有吸引力。

这并不意味着任何时候都要求具备超低延迟的能力，事实上，很少情况下才有这种需求，尽管这种情况现在正变得越来越多。一项技术首先应该满足眼前的需求，不过为未来做准备也是有好处的。现在，Flink 和 Apex 可能是目前保证低延迟且高性能的最好选择了，而 Storm 能提供中等性能水平的实时处理。

正好一次交付

很多时候都要求保证正好一次（exactly-once）处理。例如，很多金融案例，比如信用卡交易，如果不小心对一次事件进行了两次处理，结果将是很糟糕的。Spark Streaming、Flink 和 Apex 都能够保证正好一次处理机制。Storm 支持最少一次交付。通过 Trident 插件，Storm 能够实现正好一次处理，不过可能会损失部分性能。

窗口

窗口（windowing）是指流处理过程中执行聚合（aggregation）操作的时间周期。窗口的定义很多，这取决于特定的应用使用场景。基于时间的窗口将特定时间间隔内的事件分成组，适合回答这样的问题："上一分钟产生了多少次交易？" Spark Streaming、Flink 和 Apex 都具备可配置的基于时间的窗口功能。Storm 中的窗口更为朴素一些。

时间并不总是确定聚合窗口的最佳方式。例如，根据不活跃的时间长度（如 30 分钟）将网页的访问行为分成会话（session）。此时，基于时间确定窗口就不那么实用，因为不是所有会话都在同一时间结束。另一种定义窗口的方法是创建触发器（trigger），允许窗口的长度可变，但仍然要求不同聚合窗口同步（synchronization）。Flink、Apex 和 Spark 支持基于触发器的窗口技术。Flink 和 Apex 还能基于数据的内容确定窗口。

小结

一个好的流架构（streaming architecture）设计不仅仅是为了实时分析，而且对于跨系统的所有数据流（data flow）都会形成强大的优势。设计需要考虑消息传递、流处理和持久化等方面的能力。理解功能需求和期望，有利于评估可用的工具。这种基于流的通用架构设计的核心在于消息传递机制，我们将深入介绍两种目前最好的技术——Apache Kafka（第 4 章）和基于 Kafka 的 MapR Streams（第 5 章），不过在接下来的第 3 章我们先了解一下为什么流式架构能够完美支持微服务（microservices）计算风格。

第3章

流架构：微服务的理想平台

大约十年前，业界就开始倡导灵活构建大型系统，这种方式最近被称为微服务（microservices）。这种潮流最早在 Google 这样的创新型公司流行，后来很多其他公司也开始投入微服务。现在，对于很多成功的、发展迅速的公司，包括 Amazon、LinkedIn 和 Netflix，微服务方法更像是一种规则而不是异类，至少采用这种架构风格的公司都发展更快、竞争力更强。

> Netflix 应用不是一个庞大的巨型应用，每次变更都依靠中心化协调；而是一系列微服务，每一个微服务独立变更。[注1]
>
> —Yevgeniy Sverdlik

微服务的思想很简单：构建大型系统的方法应该是，将系统功能分解成相对简单的单一功能服务，这些服务通过轻量级的、简单的技术进行通信。这些服务可以由小型、高效的团队进行构建和维护。

注 1　Netflix 关闭了数据中心基础设施，参见网址 http://bit.ly/netflix-microservices。

为什么需要微服务

微服务是构建大型系统的重要趋势，主要原因在于微服务之间的弱耦合性能够实现敏捷（agility），即使是大型组织也不例外。下面我们解释原理。

微服务背后的思想最初源自面向服务的架构（SOA）以及企业服务总线（ESB）。SOA 和 ESB 起源于早期大型计算机的发展，有着严格的层级结构，每个部分负责系统的不同方面。随着万维网的发展，这种层级结构演变成一个 *n* 层架构，其中每一层由具备专业技能的团队构建。随着这些层次型系统越来越庞大，常常要将它们分解成封闭系统，首先依据业务单元，然后基于它们在业务单元内部的大致功能进行分解。

这个层次型封闭系统如图 3-1 所示。

SOA 试图校准各个封闭系统的功能边界，并隔离它们的内部细节。但是，ESB 的复杂性和严格性不仅导致服务之间的相互耦合，而且需要协调变更，这就增加了系统建设的整体难度。结果，系统很难构建和维护，一旦系统的某个部分发生变化，通常其他部分也需要改变。由于 ESB 的复杂性，相比总投入来说性能不高。这些问题使得 SOA 在实践上并没有理论上所说的那么有效。

高度可扩展、更加灵活的系统（比如基于 Hadoop 平台的系统），通过一种成本有效的方式将大规模数据变得中心化，提供给多租户使用。而且，还能添加新的数据源，传统系统的数据准备工作

也更为有效而且低成本。以上只是简单地通过收集数据来打破某些封闭性，但并没有解决单个服务是否应该独立性更强、与其他服务耦合性更低，或者大型系统是否应该以敏捷的方式构建等问题。这些就是微服务方法的用武之地了。

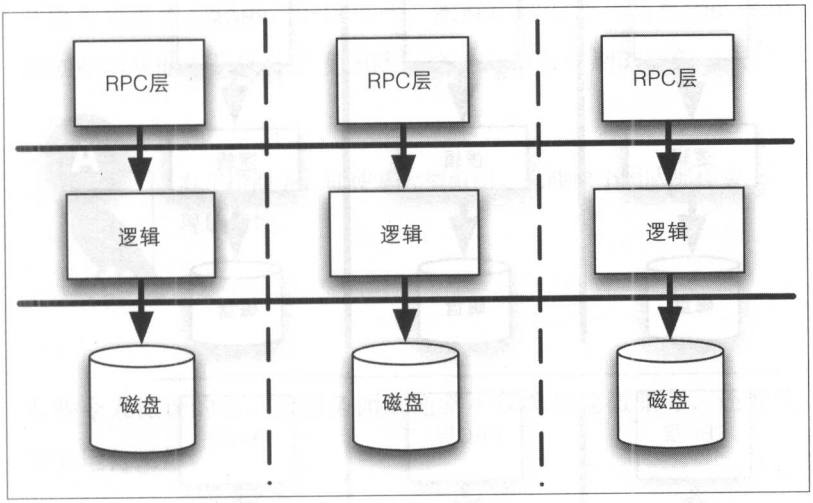


图3-1 大型系统的传统组织结构：基于技能集横向分割的组件，如图所示的横线。然后形成封闭系统，复杂性增加，即图示的垂直虚线。各个层的团队不是对单个系统负责，而是横跨多个封闭系统。这种组织结构和架构严重损害了系统之间的隔离性。

微服务设计需要专注的团队，团队内部要实现技能共享。

其结果是，系统各层之间不是强分离的，而是跨功能型团队紧紧围绕着服务开展工作。这样，系统相互之间是完全隔离的，但每个系统内部的各层不是。如图 3-2 中的 A 部分所示。

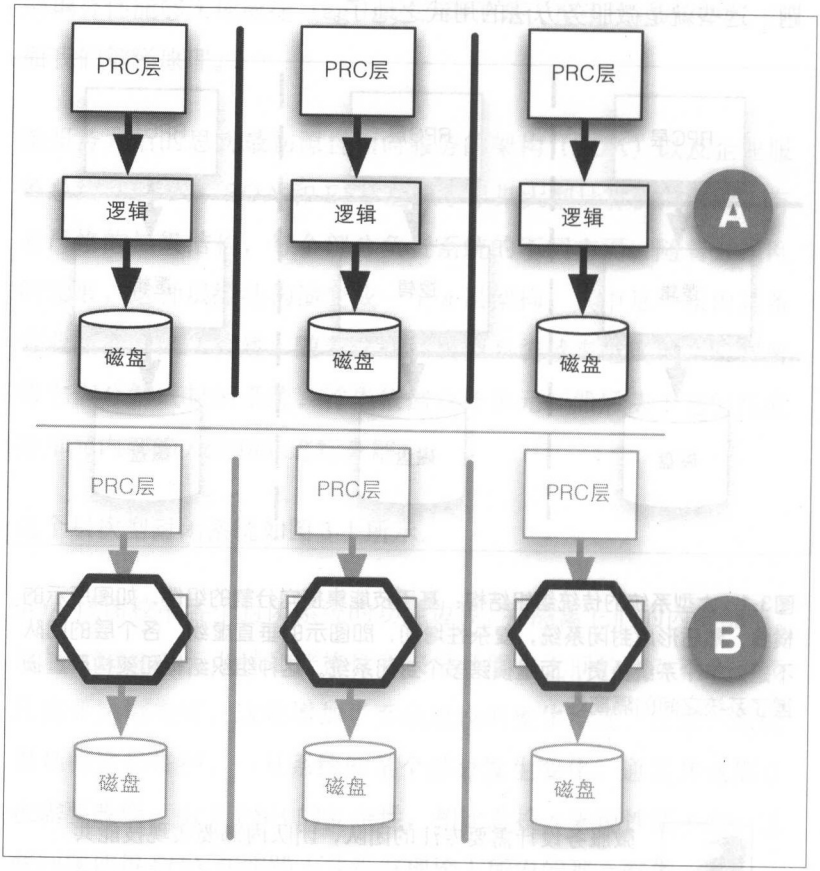


图3-2 依据大型系统的功能划分，微服务打破了横向分层。我们将此图的上半部分即A部分与图3-1（旧式结构）作对比。这里的垂直层中包含功能交叉的组，每个组代表一个服务。在该图的下半部分即B部分中，我们用一个透明的六边形代表每个微服务对应的小团队，表示不必关心服务的内部细节。

微服务方法涉及小型服务的创建，而且这些服务只能通过定义好的接口以某种有限的方式进行交互。与传统层次型方法对技能组进行划分的方式不同，微服务打破了这种横向壁垒，使得团队之间不再需要协商和相互牵制即可完成工作。显然，这种方式让每个服务变得更加简单，不仅如此，这种小型的跨功能型团队更像是创业团队而不是软件开发团队，这也会带来长期的好处。

有了微服务，即使是大型组织，也能像小团队那样实现敏捷性。

有些令人惊讶的是，微服务团队通常不仅要构建微服务，还要负责其运营工作。

微服务需要哪些支撑

为了完成特定跨功能型团队的目标，微服务之间需要进行有效的沟通，而这种交互需要是轻量级的、灵活的，目的是给每个团队分配一个作业及实现方法，然后就不管了，如图 3-3 那样。关键思想在于，服务是透明的，通过轻量的、灵活的协议与其他少数几个服务进行通信。这就牵涉到远程过程调用（RPC）协议，比如 REST 或类似 Apache Kafka、MapR Streams 这样的消息传递系统。数据格式应该能够兼容 JSON、Avro、Protobuf、Thrift 或类似的弹性系统，以满足未来的通信需求。

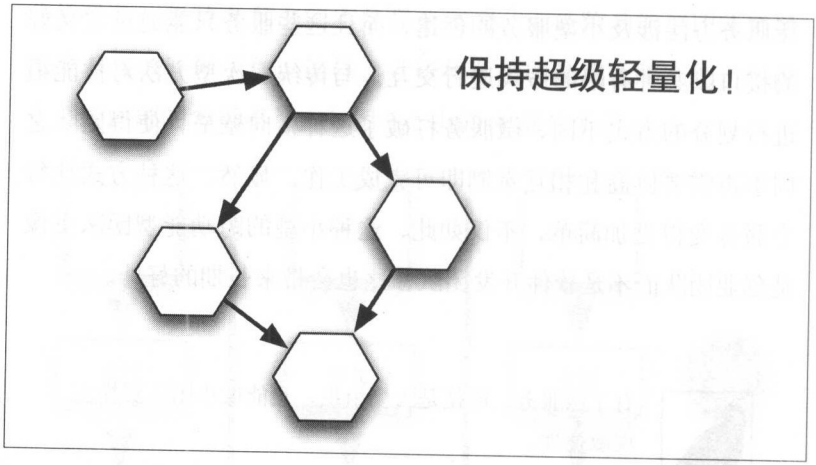


图3-3　微服务之间需要通信，这种通信方式应当是轻量级的，可以使用REST API或Apache Kafka/MapR Streams。

伴随着微服务思想而发展起来的还有DevOps（即构建服务的团队同时负责运维）、容器系统如Docker（更易于部署服务的自主版本）、持续集成（快速、经常部署新版本服务），以及REST接口（便于构建调用－响应服务）。

有了良好的流架构，这种微服务形式的计算就更易于实现。

关于微服务的更多详情

很多关于微服务的讨论起源于微服务早期在创建复杂网站方面的应用，比如Netflix。因此，很多时候我们假设服务之间的交互使用REST这样的RPC机制，涉及调用和即刻响应。实际上，微服务架构中的服务交互可能同时用到调用－响应（如REST）的同

步方法以及消息传递这样的异步方法。同步交互倾向于为用户就近选择站点。如果系统后台更倾向于分析型，即吞吐量比响应时间重要得多，而且期望的分析结果是大量记录的聚合值，那么就更需要异步交互方式。通常，物联网数据应用都是异步服务交互。

Martin Fowler 和 James Lewis 等作者强调，在微服务架构中，服务之间的数据传输应当是非常轻量级的（*http://bit.ly/smart-end-points-dumb-pipes*）。微服务架构摒弃了企业服务总线这种涉及大量转换和调度的冗繁方式，而是采用所谓笨管道（dumb pipe），仅用于传输数据。实际上，轻量（lightweight）一词应该被解释为自我服务型（self-service）的团队采用的普遍存在的（ubiquitous）机制。

在谈及微服务架构时，大部分时候人们都在强调同步这件事，但是异步也应给予足够的重视。实际上，即使某个服务请求需要立即返回结果，通常也会同时涉及两种处理方式：为了响应当前请求，即刻完成某些工作，其他延迟性工作被放入消息队列等待方便时再处理。这种延迟处理不仅提供了更敏捷的用户体验，同时也解耦了系统不同组件之间的工作调度。

流 vs. 状态

从大规模巨型应用设计理念（即基于大型数据库的事务性更新和状态访问）转变到流式微服务架构（streaming microservice architecture），是一件很难的事情。

转变的难点主要在于思维上的转变——认为计算机程序是面向状态的还是流式的。我们习惯上认为计算是一个全局状态更新的过程，这种系统的规模很难扩展（scale），而且通常这种系统需要纵向扩展（scale up）而不是横向扩展（scale out），因为这样代价非常高昂。

另一种不太普遍的观点是将计算视为独立过程元素之间的数据流动。这样会带来很多好处，最重要的是，对这种系统进行横向扩展更加轻而易举。

以上两种系统的关键区别在于，基于状态的计算要求任意时刻都保持全局一致（globally consistent）的完美状态值。举个简单的例子，假如我们想要知道地球上不同地点指定时刻的室外温度。其实我们无法在同一时刻知道所有位置的精确温度，因为即使是光速也只有每毫秒 300 千米。实际上，如果我们要仔细检查这些温度值的可靠性和科学性，需要一个月甚至更久的时间。也就是说，我们无法构造出一个能够知道全部位置当前温度的计算机程序。不过我们可以写一个程序计算局部的当前温度，然后将其他位置的温度值延迟传递（delay）过来。

同理，现代计算机无法真正拥有这种全局状态同时又不拖慢计算速度，但是，如果系统处于一定的规模和特定的时间尺度，这种状态可以近似存在。在这个规模和时间尺度之外，维持这种全局状态都是不可能的（至少是代价极其高昂的）。

> 基于流的计算摒弃了全局状态的概念，各部件不必假装知道系统其他部件到底在做什么。因此，不需要协调就可以完成更多的工作。

本书重点讨论流架构，即异步服务交互。我们还会特别讨论消息传递技术（如 Apache Kafka）及其处理系统（如 Apache Spark、Apache Flink）的进展，以及构建高级流系统的其他方面。

不过，在讨论实现微服务所需的消息传递技术之前，我们再进一步了解一下如何设计或构建流架构系统。

设计流架构：以在线视频服务为例

关于如何构建流架构，尤其是如何从微服务的角度构建流架构，这里以作者几年前创建的一个用户创造内容的（user-generated）视频网站为例，以实战经验现身说法如何使用现代数据流工具和灵活的 NoSQL 数据库建设网站。

系统的基本思想是，视频文件由用户上传，经过处理后供其他网站访问者观看。这些处理包括：提取视频的元数据，如大小、长度、原始编码、视频分辨率、视频上传的日期和时间等；从视频中提取缩略图图像（thumbnail image）；创建不同版本的视频，满足不同设备不同比特率和分辨率的需求。几年前的原系统设计如图 3-4 所示。

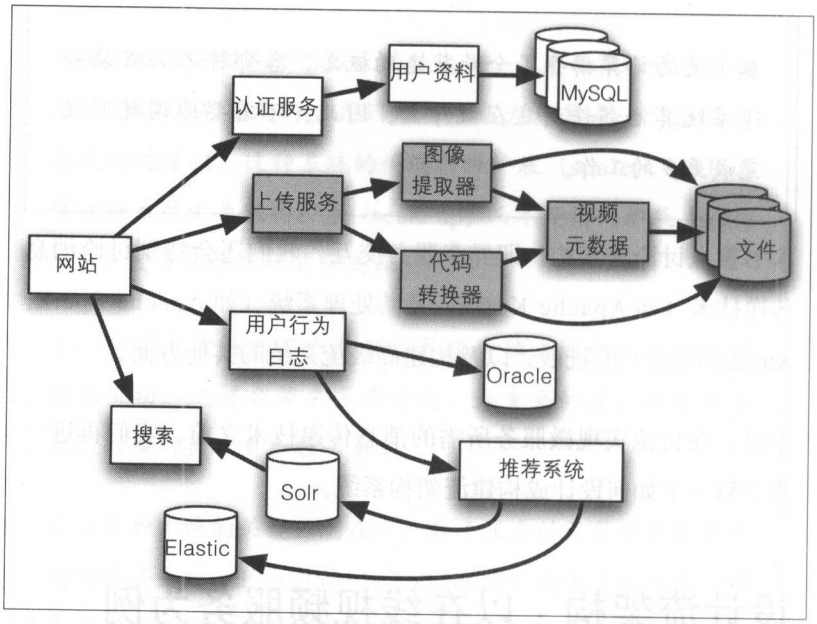


图3-4　用户生成的在线视频服务原始设计图。箭头表示手工编写的自定义数据传输，连接不同的组件或应用。通常，这些连接涉及很多不同的技术（甚至是在一个系统内部），这些数据交换在数据源和数据接收者之间的时机和握手（handshake）方面做了假设。箭头连接的组件之间存在强依赖性。灰色部分表示最难构建、难以转变成微服务系统的部分，详见图3-5。

这个旧式系统很难构建。有一个问题是，连接不同组件或过程的每个箭头，其代码实现都是唯一而特殊的，不同箭头采用的技术各不相同。这种传统的方法导致组件之间相互的依赖性：如果箭头的一端做变更，或是在另一端添加一个新的过程，都会带来更多级联的变化。系统虽然成功建成了，但成本不低，缺乏对新想法或市场变化的敏捷响应能力。

新设计：支持消息传递的基础设施

我们以事后诸葛亮的视角重新审视一下这个视频系统，利用微服务重新设计。通过分布式文件、NoSQL 数据库，以及支持服务间通信的基础设施，来改善系统的实现。

记住，微服务的一个主要目标是提供敏捷性，简化开发。这就要求将服务实现的具体细节隔离开，为此需要使用一种可持续的、高性能的消息传递技术（如 Apache Kafka 或 MapR Streams）来完成组件之间的连接，如图 3-5 所示。

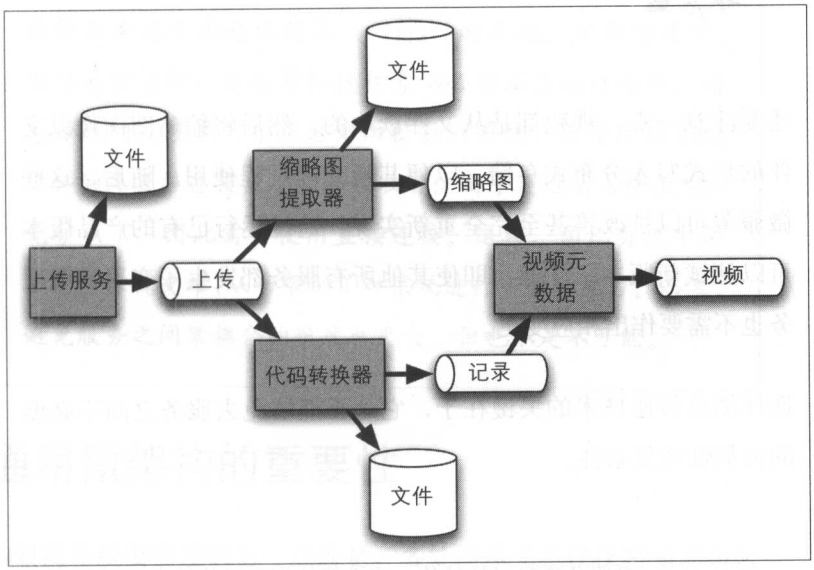


图3-5 图中以水平圆管表示Apache Kafka或MapR Streams这样的消息传递技术，标记为处理的流数据对象。这些消息流形成了微服务之间的连接，图中的灰色部分就是微服务（参照对比图3-4的旧式设计）。

开发者或架构师接触这类系统时有几个不太明显但又重要的问题需要注意。如图 3-5，如果我们只关注缩略图提取服务，我们将看到记录从“上传”流读取，写入“缩略图”流。其实这些标记还暗示了流中包含什么数据，就像变量名一样。新的微服务很容易接入，向流提供数据或者消费流的数据。因此，这种架构设计提供了敏捷性。

小团队能够精准快速地以敏捷的方式实现微服务，因为他们范围明确、接口受限。

还要注意一点，视频都是从文件读取的，然后将缩略图图像以文件的形式写入分布式存储，以便其他处理过程使用。随后，这些微服务可以被改善甚至完全重新实现。通过运行已有的产品版本可以测试新版本。注意，即使其他所有服务都发生了变更，微服务也不需要作出相应变更。

选择消息传递技术的关键在于，它是否能够免去服务之间不必要的依赖性或复杂性。

微服务需要可持续的、高性能的消息通信技术实现相互连接，同时解耦它们之间的依赖性。

通过异步通信技术连接微服务，与通过同步调用方式连接服务相比，两者所需的基础设施有很大的不同。具体来说，异步消息传递不保证在发送或接收消息的时候，发送者和接收者都正在运行（第 2 章）。实际上，如果发送者和接收者都不在运行，也可能存在消息的发送和接收。这就是说，我们需要在基础设施层保障异步消息传递，不能指望发送者或接收者来处理消息。如果没有基础设施提供支撑，就会导致发送者和接收者的实现存在耦合性。

超低延迟需要另一种方法

我们来考虑要求超低延迟（<1ms）的系统。这种情况下，当消息经过时，发送者和接收者都必须正在运行当中。消息传递的处理不同于延迟需求松散的系统，而是使用 0mq 这样的库直接连接发送者和接收者。由于延迟需求降至 100 毫秒以下，因此需要使用直接连接。这样，向已有流中添加消费者就更难，从而限制了对系统进行审查的能力。而且，避免服务之间紧耦合的难度也更大，但也不是不可能。

通用微架构的重要性

构建流系统要注意的另一点就是，各处理单元不仅仅需要那些显而易见的输入和输出。以缩略图图像提取为例，缩略图提取器还应当将常规操作信息，如处理的视频数量、处理时间直方图等发送到度量数据流（metrics stream），将描述异常处理的记录发送到异常数据流（exception stream）。如图 3-6 所示。

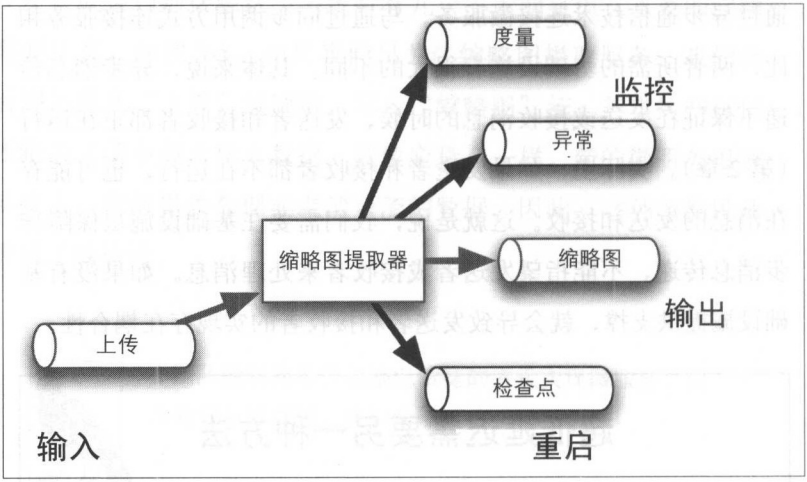


图3-6　处理单元应该基于通用微架构，通过流输出度量记录、发出异常信号。有状态的处理需要输出检查点流，在重启时形成重新装载（reload）状态，并与某个输入消息位置状态相关联，保证处理工作正确重启。

度量

就最佳实践来说，整个流系统中的组件应当以消息流（message stream）作为收集度量和异常的方式。任何不值得监控的服务都不应该运行。

通用微架构（universal microarchitecture）就是用来收集诸如度量和异常这样的信息的。

命名问题

在这种架构中，所有数据传输的命名似乎都有些奇怪。从传统意

义上说，如果用这种方式画系统框图的话，只要用箭头就可以了，通常不需要命名。每个箭头的具体实现取决于源（source）和消费者（consumer）之间的约定。一般来说，几乎每个箭头的约定都不太一样，这种不一致性导致生产者（producer）和消费者（consumer）之间存在大量耦合。这种耦合性违反了微服务的核心价值观，最终导致系统像传统架构那样难以修改。

正确的做法是以异步消息传递提供基础设施能力，服务之间的所有消息都采用统一的机制进行传递。对连接进行命名，表示新的消费者到消息流的连接。图 3-7 将消息传递基础设施简化为一个箭头和一个消息内容标签。实际上，箭头也是架构的重要组件，本质上它的作用是后台不可见的部分。

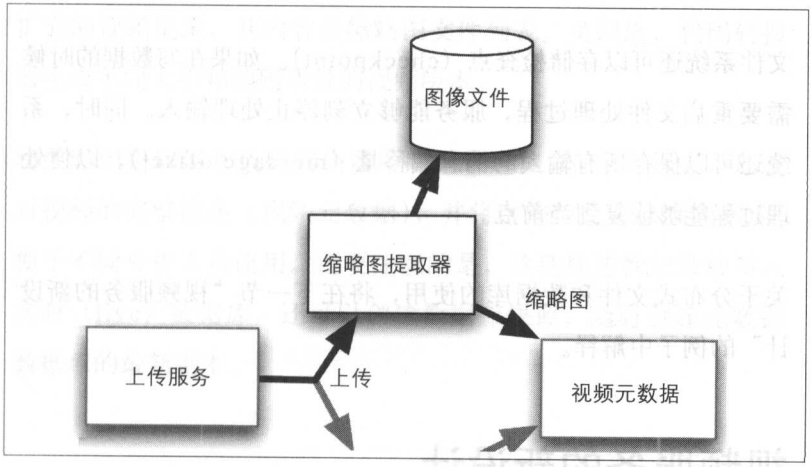


图3-7　缩略图提取器服务的输入和输出是隔离的。出于简化的目的，为了突出非基础设施的部分，我们以带标签的箭头（而不是图3-5和图3-6中那样的水平圆管）表示消息流。注意，上传（upload）消息流是指原始视频文件，而输出的缩略图（thumb）消息流则是缩略图提取过程提取出来的图像文件。

为什么使用分布式文件和 NoSQL 数据库

如果所有组件之间的连接都采用流（stream）的形式，并尽可能地使状态局部化（local），这似乎与将缩略图图像文件保存到分布式文件系统相矛盾。理论上说，虽然通过消息队列可以推送任何类型的数据，但是很多时候文件仍然不失为一个很好的解决方案。如果有合适的分布式文件系统，那么大型数据对象（超过若干 MB）的持久化就更加容易。假设数据以文件形式存储，我们有很多工具用于处理这些对象（如图像提取器），或者将它们提供给用户（如 Web 服务器）。此外，Kafka 存储消息的默认最大值只有 1MB，MapR Streams 最大存储 2GB。这些上限都不够大，我们很难保证不需要一个更大的文件。

文件系统还可以存储检查点（checkpoint）。如果在写数据的时候需要重启文件处理过程，服务能够立刻停止处理输入。同时，系统还可以保存所有输入的消息偏移量（message offset），以便处理过程能够恢复到当前点。

关于分布式文件和数据库的使用，将在下一节“视频服务的新设计”的例子中解释。

视频服务的新设计

将上述所有改进综合起来，我们就得到了视频案例的新设计。具体变化包括：

- 通用微服务架构设计。
- 服务之间通用的可持续、高性能的消息传递技术（如 Apache Kafka 或 MapR Streams）完成轻量级的通信，维持微服务之间的独立性。
- 使用分布式文件系统、NoSQL 数据库以及快照。

新设计采用了微服务和流的方法，架构中每个处理单元的目的都很明确，几句话就可以描述清楚，如图 3-8 所示。更重要的是，每个处理单元的原型只需要几个小时即可实现。

例如，文件上传服务将原始视频存储为分布式文件系统中的文件，并发送一条记录，包含标题以及视频数据文件名称。缩略图提取器读取这些记录并处理视频文件，生成很多图像文件，以及一条扩充的视频记录，其内容是缩略图文件列表。类似地，代码转换器生成不同大小和编码质量的视频版本。

将缩略图提取器和代码转换器中的记录连接（join）起来，形成对视频的完整描述（视频元数据），并将这些信息存储到数据库，便于不同分析人员使用。还要注意的是，这些描述性记录被写入实时（live）数据库，还可以创建数据库快照，随时创建元数据数据库的最新副本。

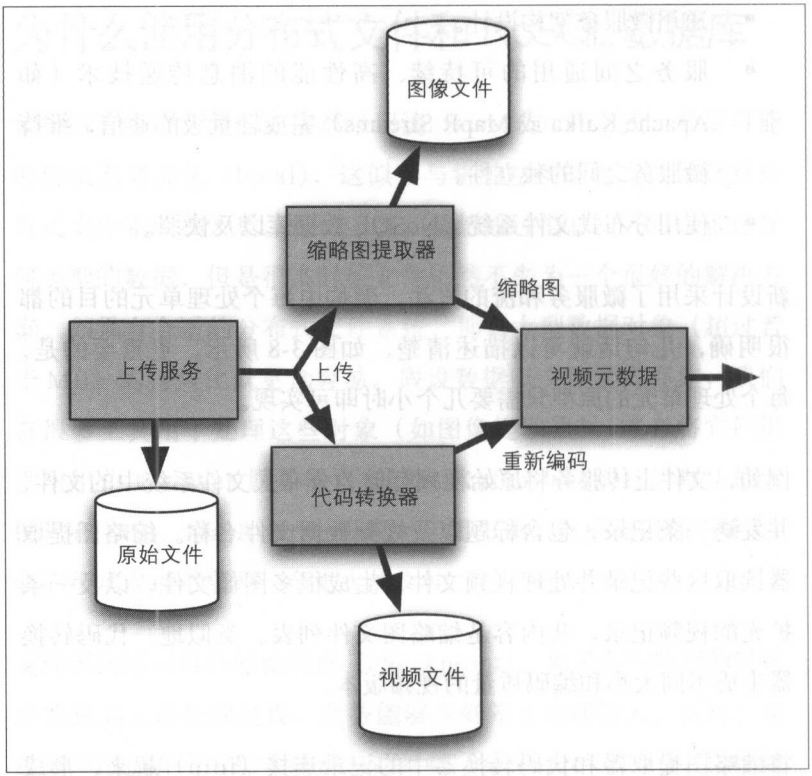


图3-8　在线视频服务的新设计。这个例子体现了我们前面提到的所有的思想。处理单元以矩形框表示，文件或数据库持久化以圆柱体表示。为了简化问题，通过通用消息传递技术实现微服务之间的连接，以标记有流内容的箭头表示，不再使用图3-5和图3-6那样的管状符号。注意，这是一幅极其简化的图，没有画出度量流。与旧式设计共有的组件以灰色框表示。

小结：综合平台视角

为了确保微服务流架构顺利工作，其支撑平台具备以下约束条件。

首先，所有的服务都必须使用一致的（consistent）、普适的（ubiquitous）流数据传输机制。这与同步方式的微服务不同，同步请求暗示服务本身可以接收查询和提供响应。也就是说，同步微服务不需要基础设施支撑，只需要一个可靠的网络连接即可，但这种方法限制了灵活性，需要更多的管理性开销。

因此，我们推荐异步流服务。在这种流基础设施方法下，当消息被发送到某个流时，接收方服务不必正在运行。同时这也要求流本身能够持久化消息，直到消息被读取和处理。

另一个关键要求是，服务的新实例（instance）需要装载由该服务维护的任何状态。这通常意味着，在新实例准备运行的时候，需要重新读取以前的消息，而且要与已经读取过它们的原始实例保持一致。此外，对服务进行调试或取证检验，也要求对以前的消息进行检查。

目前满足这些需求的最佳办法是，在整个流系统中使用一种可重放的（replayable）持久化消息系统，如 Kafka 或 MapR Streams。

很多系统，比如本章提到的视频处理链，需要操纵大型对象，规模从几百 MB 到几 GB 不等。虽然将非常庞大的消息写入消息系统也是可以的，但通常不是一个好办法。通过文件式 API（如打开、读、写、关闭操作等）进行读写来实现这种大型操作更加妥当。另一方面，消息传递 API 通常要求在单个调用中传递整个消息。也就是说，超过几 MB 的对象可能需要采用其他方法传递，比如分布式文件系统。视频处理链将视频文件、缩略图图像和转换过

的视频写入文件，然后将这些文件的引用（reference）作为消息进行传递。

综上所述，成功实现流架构通常需要Kafka式的消息传递及分布式文件系统，为所有服务提供实用工具。这种综合系统支持消息、文件和表，更易于构建和维护异步微服务。

第 4 章

使用Kafka进行流传输

第 2 章已经讲述流架构设计的核心在于消息传递能力，以满足大规模系统的基本需求。我们推荐两个技术：Apache Kafka 和 MapR Streams。这一章我们将详细讨论 Kafka，它是流架构消息传递的先驱。

Kafka 的动机

Apache Kafka 最初是 LinkedIn 的一个工程项目，旨在规范化服务之间的数据迁移。在 LinkedIn，很多服务最初都采用关系型数据库，Java 过程之间通信都使用远程方法调用（Remote Method Invocation，RMI）。

不幸的是，这两种方式很难应对服务数量的迅速扩张，以及迁移数据量的急剧增加。一旦某个服务需要与另一个服务通信，就需要开发和维护一个转接器（adapter）。更有甚者，每个转接器都要

针对发送者和接收者做适当修改，每个通信服务对或多或少都会互相暴露一些实现细节。因此，更新系统极其困难，而且在服务之间迁移这么多信息也十分困难。

一直以来，SOAP、CORBA、Java RMI 等系统都基于这样的假设：严格版本控制和严格类型安全是进程间通信的关键。但是 LinkedIn 等团队十几年的经验完全不是这样，他们遇到的难题是：如果服务在通信的时候使用任何强类型约定进行交互（如严格版本的 API 或数据库模式），那么每个服务相当于对其他服务进行二次开发或者修改。用不了多久，这些修改就会比服务本身还繁重。这些依赖性是一种负担。

LinkedIn 的工程师们认识到这种窘境之后，认为服务之间需要一种一致性的通信机制，来解决他们面临的种种问题。显然，通信必须是异步的、基于消息的。为了解耦消息的发送者和接收者，需要持久化所有消息，这一点很重要。不过，持久化对吞吐量有所要求，传统的消息传递系统难以支撑前面提及的微服务。

Kafka 的创新

Kafka 采用了传统的消息队列思想。生产者（producer）向消息队列（或主题，topic）发送消息，该消息标识为主题名称。消费者（consumer）从主题中读取消息，如果消费者订阅的主题有了新的消息，则通知消费者。

但是，Kafka 与以前的消息传递系统存在一些重要区别。这些重要的技术创新使 Kafka 能够为大规模服务架构解决消息传递层的问题。

Kafka 的重要创新包括：

要求按照顺序确认（acknowledged）所有消息。

不再需要为每个监听者（listener）跟踪每个消息的确认情况，读消息的操作与读取文件的操作类似。

设置消息持久化的时间（单位：天甚至星期）。

不再需要跟踪特定消息的读取情况，设定消息的保留时间，确保消息被读取之后才能删除。

要求消费者管理即将处理的下一条消息的偏移量（offset）。

Kafka 存储已提交（commit）消息的实际偏移量（使用 Apache Zookeeper），互相独立的消费者的偏移量也是不相关的。应用程序甚至完全可以在 Kafka 之外管理这些偏移量。

这些创新带来的技术层面的影响是，Kafka 能够将消息写入文件系统。按照消息产生的顺序依次写入文件，然后按照消费（consume）的顺序依次读取消息。这种设计意味着 Kafka 消息代理（broker）极少出现非顺序性读写文件，因此 Kafka 处理消息的速度非常快。

Kafka 的基本概念

在我们详细讨论如何使用 Kafka 之前，首先了解一下 Kafka 系统及其作用。总的来说，Kafka 向用户展示了一个特别简单的模型，不过其中的术语可能比较陌生。

我们来看看 Kafka 系统的术语。*生产者*（producer）将消息发送给 Kafka *代理*（broker），即 Kafka 集群中的一个服务器。这些消息由消费者（consumer）读取。图 4-1 描绘了这个结构。

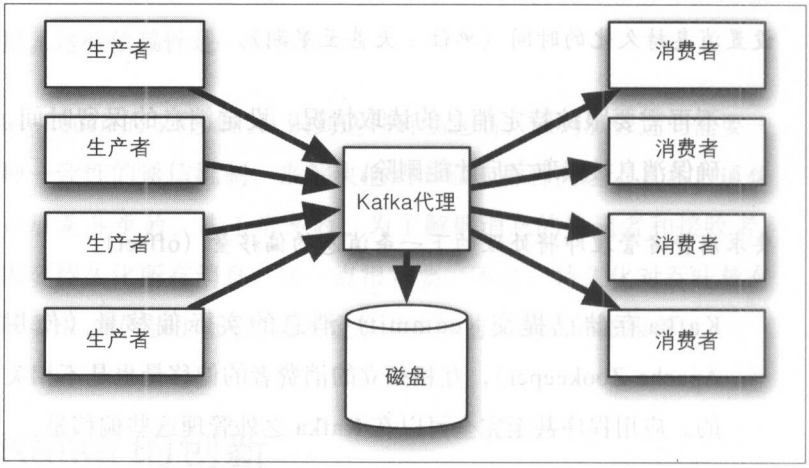


图4-1　生产者将消息发送给Kafka代理，然后由消费者读取。代理将消息持久化到磁盘上的文件系统，保证可靠性，确保消费者能够接收到来自代理的消息。

Kafka 代理负责管理传输中的消息。*消息*（message）的内容是应用程序自定义的序列化格式的字节，与某个*主题*（topic）的生产者相关联，其中主题是指消息组的高级抽象。

通过主题，消息的消费者不需要读取很多不相关的消息，就能找到相关消息。

代理存储和转发多个主题的消息，多个生产者将消息发送到一个主题。生产者在将消息发送到 Kafka 代理之前缓存大量消息。通过设置缓存消息的数量或消息逗留的时间，生产者可以控制缓存多少消息。

排序

最后，消费者读取这些消息。最简单的情况是，一个主题的所有消息都被一个消费者读取，按照代理接收这些消息的顺序依次读取。如果要求消息处理的吞吐量更高，可以将主题分割为多个分区(partition)。然后，消费者以多线程的方式从这个主题读取消息，但这时只能保证单个分区内的消息有序，而且线程的数目不超过分区的数目。生产者直接指定消息分配到哪个分区，或者通过关键字的哈希值间接确定消息的分区分配情况。

重要的是，某个主题分区内的消息是有序的，并且消费者不保证每个消息的确认 (acknowledge) 是有序的。消费者的读取点 (read point) 确定了下一条要读取的消息是哪条。默认情况下，在读取点之前读取消息时，读取点可以是任何一条消息的起始位置，也可以是代理最早或最晚收到的消息的起始位置。读取点之后的消

息是按顺序被读取的，直到读取点被显式重置。这种文件式 API 与传统的消息队列式 API 完全不同，在消息队列式 API 中，消息读取和确认的顺序是任意的。

持久化

Kafka 和传统的消息系统还有一个重要区别，即消息的持久化是无条件的。而且，基于主题（topic）的保存策略按照接收消息的顺序保存或丢弃这些消息，而不关心消息的消费者是谁。

有一种例外情况，即删除主题中的旧消息其实是对这些消息进行压缩（compact）。如果相同关键字（key）不再产生新消息，压缩可以保存消息，而删除不保存消息。压缩的目的是将主题的更新保存到键值（key-value）数据库，而不会造成无限制的数据增长。如果一个键更新了很多次，则只考虑最后一次更新，因为前面的更新都会被最后一次更新覆盖。有了压缩以后，主题的规模不会远远超过当前更新的数据表，此外，由于对主题的全部访问都是按照时间顺序的，存储主题的方法可以非常简单。

Kafka API

Kafka API 经历了巨大演变。最初，API 非常简单，复杂性都被转嫁给了使用 Kafka 的程序。后来，低级和高级 API 被研发出来，不过两者之间界限不够明确，常常需要同时使用高级和低级 API 才能完成一些标准型任务。

最近发布了 Kafka 0.9 版，将低级和高级 API 合并成一个 API，简化了客户端程序编写。这样，Kafka 的用法更加直观。如果可以的话，所有新应用都应该使用 0.9 版 API（或者已发布的最新版本）。

简单的 Kafka 程序

这一节介绍 Kafka 0.9 API 的基本框架，以及性能调优方面的部分知识，但我们不具体讨论如何编写 Kafka 应用程序。我们写了一些程序示例，GitHub 上可以下载。关于如何使用 Kafka 0.9 API 编写代码，博客上提供了详细的例子，并给出了 GitHub 代码库链接，地址为 *http://bit.ly/apache-kafka-code-samples*。此外，还可以去 Apache Kafka 官网查看 API 文档。

KafkaProducer API

在 Kafka 中，`KafkaProducer` 通过代理将所有消息发送给消费者。发送完毕以后，发送进程将消息分配到主题。接着，通过消息对应的键（key）的哈希码，显式或隐式地将消息分配到主题内的分区。

每个 `KafkaProducer` 实例（instance）表示一个到 Kafka 代理的单独连接，不过通常很少存在多个实例，或者说多实例几乎没有速度优势。`KafkaProducer` 是线程安全的（thread-safe），因此在多线程中使用单实例不需要进行特殊处理。

在Kafka中，所有消息都是异步发送的，即消息交给KafkaProducer准备发送之后的一段时间以后，消息才真正通过网络发送给代理。等到缓冲区满（Kafka的buffer.size设置）或指定时间段（Kafka的linger.ms设置）之后，消息才会真的发送。默认情况下，缓冲大小和超时的值都很小，这会影响吞吐量，不过延迟情况还好。

数据从生产者传输到Kafka，存留（durability）情况各不相同，取决于不同的设置。主要由两个参数控制：acks（生产者配置）和min.insync.replicas（主题级别的配置）。在最底层，只需要在确认（acknowledge）之前发送消息即可。再往上层一点，要求至少一个代理确认消息。在最高层，要求所有拥有主题最新副本的代理都必须确认收到消息。

通常来说，我们强烈建议使用设置min.insync.replicas=2和acks=all。即，所有主题最新副本的代理都要确认所有消息，而且至少有两个代理确认了所有消息。这种策略类似于MapR文件系统，确保即便只剩一个最新副本，也不会导致数据丢失。而且，不会出现单节点或磁盘故障，不会丢失已确认的消息。

KafkaProducer有很多与性能有关的属性。大多数属性的默认设置能够提供较低延迟，某些参数稍做调整就能极大地提升性能。例如，缓存更多记录（batch.size参数），等待一段时间之后再将这些缓存记录发送到代理（linger.ms参数）。对吞吐量敏感型

(throughput-sensitive) 应用来说，将以上两个参数分别设置为 1 MB 和 10 毫秒，就能带来很大的性能影响。例如，假设创建 100 万条记录基准（benchmark）所需时间为 4 秒，如果使用默认参数，创建并发送这些记录需要 24 秒。将 `batch.size` 和 `linger.ms` 修改为上述推荐值后，运行时间降至 8 秒，吞吐量提升了约 5 倍。如果继续调高参数值，就基本上没什么作用了。实际应用的结果可能不太一样，不过显然更多的缓存和更长的等待时间，对性能的影响很大。

有了 `KafkaProducer` 对象以后，就可以通过 `send` 和 `flush` 方法向代理发送消息。`send` 方法先将消息复制到一个内部缓存，然后根据 `KafkaProducer` 的策略，或者显式调用 `flush` 的时候，自动将消息发送给代理。注意，由于 `send` 方法是完全异步的，因此其返回结果不可能是尝试将消息发送到代理，而是返回一个等待发生的未来事件，有的版本还可以返回待发送消息的回调函数(callback)。不论是返回未来事件还是回调函数，都可以确定消息是否已经成功发送。

前面提到过，发送到代理的每条消息最后都会被分配到主题的一个分区。发送消息的生产者需要确定如何基于 `send` 方法选择分区。通过分区号或键值的哈希码或轮询调度的方式，直接或间接地分配分区。

如果向代理发送的数据非常多，调用 `flush` 可能没有用，因为发送的消息量太大，任何情况下数据都会很快刷新。另一方面，如果发送的消息太少，显式调用 `flush` 可能会增加延迟。此外，`flush`

还有助于在发送其他消息之前，确定消息是否已经到达代理。在 GitHub 项目（*https://github.com/mapr-demos/kafka-sample-programs*）中，com.mapr.examples.Producer 类中实现了一个简单的消息生产者。

KafkaConsumer API

从很多方面来说，Kafka 消费者端比生产者端的代码实现难度更大。主要问题包括，与主题分区有关的消费者组（consumer group）概念，确定故障进程处理了哪些消息，以及消费者配置对吞吐量的影响等。

消费者组的重点是，以可控的方式混合采用两种模式：统一广播所有消息（有助于添加新的消息消费者）以及为每条消息指定单个处理程序（有助于实现消息的并行化处理）。Kafka 通过消费者组的概念调和两种极端情况。订阅某个主题的所有消费者组会收到全部消息，但在一个消费者组内部，每条消息只由一个消费者处理。

此外，Kafka 要求在生产者端将主题分成分区（partition），而不是完全不控制消息发送给哪个消费者。因此，生产者决定将消息发送到哪个主题及哪个分区，而消费者组确定究竟哪个消费者处理哪个主题分区。

生产者、主题、分区、消费者组以及消费者之间的关系如图 4-2 所示。

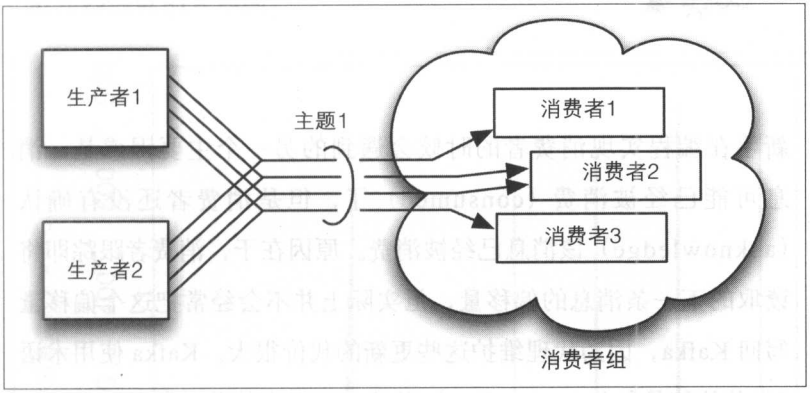


图4-2　生产者确定每条消息发送到主题的哪个分区，同时将每个分区分配给订阅该主题的每个消费者组中的一个消费者。这里，平行线表示主题，其中每条线表示主题的一个分区。将分区分配给不同的消费者，能够并行处理消息；虽然同一个分区内的消息是有序的，但是由于不同分区的消息延迟不同，排序也会受到影响。

虽然 Kafka 能够保证单个主题分区内消息的有序性，但是应用程序不应该过度依赖事务排序（transaction ordering）。例如，如图 4-2，假设来自生产者 1 的消息 *m*1 和 *m*2，与来自生产者 2 的消息 *m*3 和 *m*4，都被发送到同一个分区。*m*1 总是在 *m*2 到达之前先被消费者接收，但是很难断定 *m*1 是否一定在 *m*3 之前。这时最好采用防御式编程。如果两个生产者运行在同一个节点上，那就容易了，否则，可能需要非常高精度的时钟同步（clock synchronization）。

应对排序困惑的最佳防御方法就是高精度时钟。

新手在编程实现消费者的时候会遇到的另一个主要困惑是，消息可能已经被消费（consumed）了，但是消费者还没有确认（acknowledge）该消息已经被消费。原因在于，消费者跟踪即将读取的下一条消息的偏移量，但实际上并不会经常把这个偏移量写回 Kafka，因为代理维护这些更新的代价很大。Kafka 使用术语“当前偏移量”（current offset）以及“已提交偏移量”（committed offset）分别描述上述两个概念。如果消费者一直以有序的方式处理消息，最后通过提交当前偏移量正常退出，那么没有任何问题。大部分情况下，当前偏移量在已提交偏移量之前，这就无所谓了。

另一方面，如果消费者还没向代理提交偏移量就挂掉了，接着另一个消费者又开始处理同一个分区，那么它会从上一个已提交偏移量开始，因此有些消息可能会被处理两次（如果在批量处理消息之后提交偏移量），也可能完全不做任何处理（如果在批量处理消息之前提交偏移量）。通常来说，处理消息和提交偏移量很难构成一个原子性操作，因此几乎总是无法保证所有消息都刚好只被处理一次。

新手经常遇到的最后一个困惑是，消费者如何决定读什么数据。

在 Kafka 0.9 中，定义消费者如何决定读什么数据以及什么时候读

的默认参数设置，能够确保每秒消息量不是特别大的主题上的延迟较低。如图 4-3 所示，这个设置导致短时间内吞吐量很高，但是几秒后就陷入崩溃。

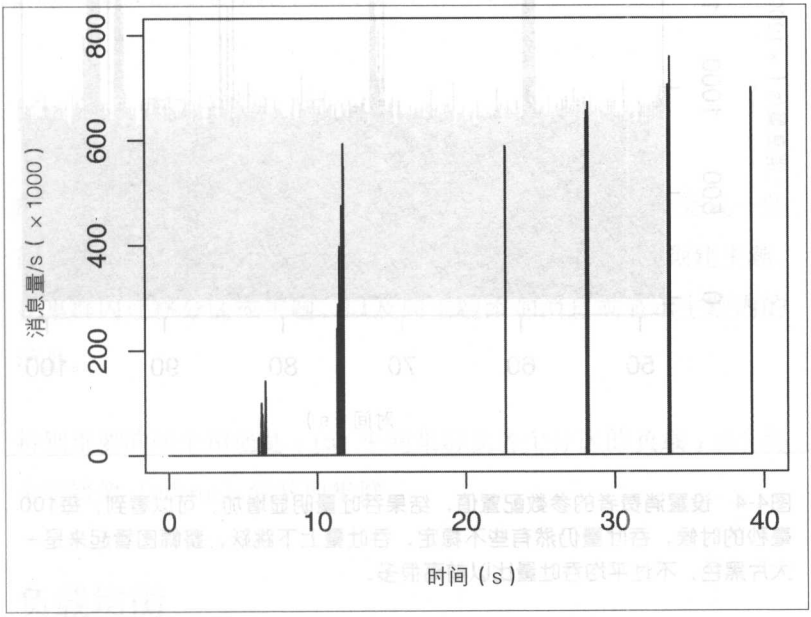


图4-3 如果采用receive.buffer.bytes默认值，吞吐率不稳定，消费者无法继续正常工作。

处理这种不稳定性的关键是将 receive.buffer.bytes 值设置得足够大，使处理过程能够持续高速率地进行下去。通常还需要增加 fetch.min.bytes 和 max.partition.fetch.bytes 的值。从图 4-4 可以看出，如果这些消费者参数值设置合理，情况就好多了。

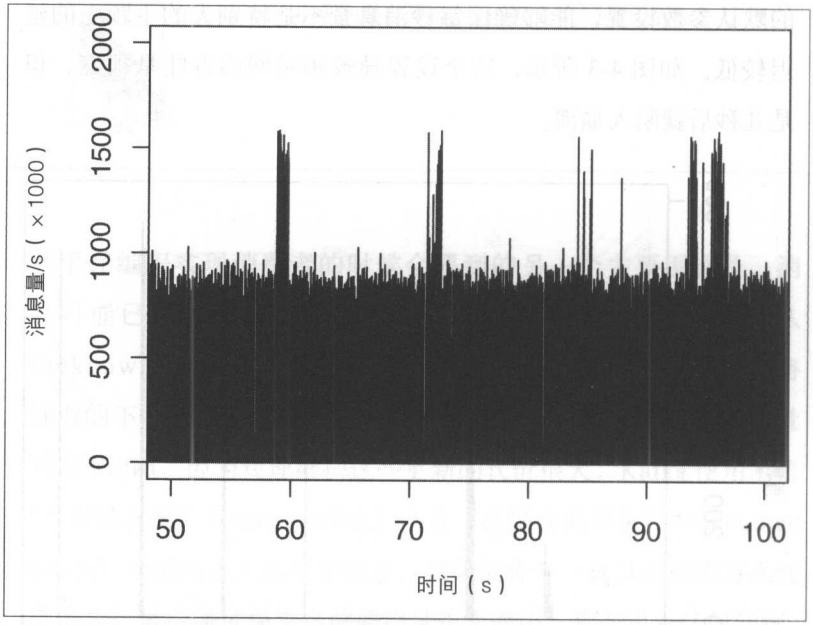


图4-4 设置消费者的参数配置值，结果吞吐量明显增加。可以看到，每100毫秒的时候，吞吐量仍然有些不稳定，吞吐量上下跳跃，整幅图看起来是一大片黑色，不过平均吞吐量比以前高很多。

遗留 API

最新发布版 Kafka 还存在一个潜在的困惑，即 Kafka 保留了以前旧的消费者 API，新 API 并不能完全替换旧 API 的功能，新旧 API 混合使用。这导致编程模型模糊不清。更重要的是，旧 API 暴露了 Kafka 的实现细节，严重影响性能的提升，难以提高可靠性和安全性。

Kafka 0.9 版保留了旧 API，但是新应用程序不应该使用这些旧 API，而应当采用新的交互方式，即通常所说的 0.9 API。我们提供的示例程序（*https://github.com/mapr-demos/kafka-sample-programs*）很好地诠释了如何使用 0.9 API。

Kafka 实用程序

Kafka 还有很多实用程序，帮助管理 Kafka 集群，以及完成一些简单的诊断工作。这些实用程序可用于启动代理进程、创建主题、在集群内迁移分区或主题，以及向主题添加消息或显示主题内的消息。

特别重要的两个用途是：(a) 平衡集群及各个分区的负载；(b) 将主题镜像（mirror）到其他集群。

负载均衡

Kafka 的模型非常简单，一个主题分区全部保存在一个代理上，可能还有若干个代理作为该分区的副本（replica）。分区不在机器之间分割。Kafka 不会自动迁移分区以平衡负载量或存储空间，这是一个很严重的问题。因此，如果向集群添加新节点，必需手动将数据转移到这些新节点上。这项工作很棘手，属于劳动密集型任务，我们很难判断哪些分区可能会增长或负载过高。

镜像

在 Kafka 集群之间制作数据镜像的实用工具是 MirrorMaker。MirrorMaker 运行的时候会启动大量线程，每个线程都订阅 Kafka 集群中的主题。多个 MirrorMaker 进程能够运行在不同的机器上，从而获得更高的并行性和容错性。每个 MirrorMaker 线程由一个消费者和一个生产者构成，其中消费者从 Kafka 源（source）集群读数据，生产者向目标（destination）集群写数据。可以存在多个源集群。MirrorMaker 利用消费者组实现线程和服务器之间的负载均衡。

然而，使用 Kafka 镜像应该注意几件事。

源集群和目标集群之间本质上不存在连接，源集群将消息上写入镜像主题，MirrorMaker 读取这个主题，然后发送给目标集群的相同主题。不保存偏移量和消费者读取的偏移量，因此镜像无法用于灾难恢复情况下消费者回滚。

Kafka 中主题镜像是树状的，而不是通常采用的图。其主要原因是，如果复制路径中存在环路，流量就会呈指数式增长，因为制作镜像的过程发现不了重复值。

Kafka 的陷阱

Kafka 以十分简单的方式实现了流架构，这是一个巨大的转变，不过这项工作仍在继续。Kafka 从底层的基础功能构建，如数据复制、容错和异地复制等，这在一定程度上阻碍了 Kafka。这种情况促使了一些挑战性功能的发展。

尽管如此，Kafka 在很短时间内取得了巨大进展，而且很显然它扮演了很重要的角色，至少是部分扮演了重要角色。Kafka 给人的第一印象特别好，的确是这样。你可以下载 Kafka，测试示例程序，示例应用运行时间不超过 10 分钟（我测试过了）。

另外，Kafka 扩展性很好，简化了大型系统的设计，使用比较方便，特别是 0.9 API。难怪 Kafka 拥有那么庞大的用户社区，而且用户数还在增加。有什么理由不喜欢它呢？

产品环境下的 Kafka

一旦 Kafka 大规模进入产品环境，就会出现很多重大问题。有些问题与 Kafka 目前的设计原理有关，短期内无法改变。其他问题则可能随着时间的推移得以改善。

主题和分区的数目有限

Kafka 集群能够处理的主题数目是有限的，达到 1000 个主题左右时，性能就开始下降。这些问题基本上都跟 Kafka 的基本实现决策有关。特别是，随着主题数目增加，代理上的随机 I/O 量急剧增加，因为每个主题分区的写操作本质上都是一个单独的文件追加（append）操作。随着分区数目增加，问题越来越严重，如果 Kafka 不接管 I/O 调度，问题很难解决。除了分区数目的有限性以外，还可能存在其他限制，而且有些限制相当严重。比如，单个进程打开文件，文件描述符的数目通常是有限的。

为了更好地适应产品环境，Kafka 需要解决文件系统设计固有的大部分问题。这可能需要好几十年的不懈努力才能解决。

请记住，主题数目限制带来的实际影响，随应用和设计的不同而不同。目前为止，这个限制对大部分 Kafka 应用而言都不是什么大问题。如果将 Kafka 作为多租户资源，不同用户都想各自设置自己的主题数目，这时这个限制就成问题了。

此外，主题数目的有限性还限制了问题空间（solution space），导致 Kafka 无法作为很多应用的主存储或归档存储。例如，对智能计量电力设施来说，合理的系统设计是每个计量都拥有单独的主题。这种设计对 Kafka 来说是具有争议的，因为在 Kafka 中上百万乃至上千万个主题是不合理的。

手动均衡分区负载

在 Kafka 中，一个分区副本（partition replica）必须在一个机器上，而不能在多台机器上分割存储。随着分区不断增长，集群中有的机器运气不好，会正好被分配几个大分区。Kafka 没有自动迁移这些分区的机制，因此你不得不自己来。监控磁盘空间，诊断引起问题的是哪个分区，然后确定一个合适的地方迁移分区，这些全都是手动管理性任务，在 Kafka 集群产品环境中不容忽视。

如果集群规模比较小，数据所需的空间较小，这种管理方式尚且奏效。但是，如果流量迅速增加或没有一流的系统管理员，那么情况就完全无法控制。更糟糕的是，任何资源竞争带来的不稳定性都会导致研发团队切换到他们自己的 Kafka 集群，从而打破流数据作为基础设施资源的整个概念。而且，分区的所有副本都必须完全存储在相应的代理上，这个要求也不利于数据的长期归档。

为此，需要对 Kafka 的架构作出很多根本性的改革。修改 Kafka 的基本实现，将分区分成多个文件，不再限制一个分区只能保存在一台机器上——这样做看起来是合理的，但也会导致集群中的代理不仅需要维护分区的主代理（master broker）的位置，还要维护每个主题分区的每一小段（segment）所在的位置，以及哪个代理是当前段的主代理。

这种管理是可实现的，但需要大量精力，具体处理细节非常多。对于负载和空间均衡的问题，MapR Streams 技术的解决方法完全不一样，我们将在第 5 章介绍它。

没有固有的序列化机制

Kafka 没有偏好的数据结构序列化机制。也就是说，不同的开发者可以选择不同的序列化方法。服务之间需要通信，其中消息的序列化方式应当是一致的，这是成功使用 Kafka 的关键。

应该基于性能还是未来发展的考虑来选择偏好的序列化机制，关于这个问题存在一些争议。就单个组织来说，这种争议往往就是传统用法的问题。如果在某个组织内部，一套序列化框架已经文化氛围浓郁、专家技术娴熟，那就很难再使用另一套更好的序列化框架了。

另一方面，所有序列化都是在 Kafka 之外实现的，即使没有必要，也会强制性地至少再一次复制数据。这极大地损害了消息系统的性能。

不论 Kafka 是否倾向于某一个序列化系统，各组织都必须有自己的强烈偏好，以避免消息传递混乱。这更多的是一个社会问题，而不是系统问题。

越早建立这种约定对后期的益处越大。

镜像的不足

Kafka 的镜像系统非常简单——对很多企业应用来说，有点过于简单了：简单地将消息转发给镜像集群，源集群的偏移量在目的集群中不再有效。也就是说，生产者和消费者不能从集群故障转移（fail over）到镜像。企业系统通常要求具备故障转移能力，否则可能完全丧失 Kafka 的优势。

在设计复制链（replication chain）的时候，Kafka 镜像设计也存在类似的困扰。由于源集群和镜像互相都不知道对方，从集群到集群的消息复制一旦存在环路，就会导致消息在镜像过程中被来回复制。接着很快就是灾难降临。

无回路、不分裂的镜像，意味着复制模式是树形的，这种设计非常脆弱。从实践上说，丢失树的任何一个连接，都会导致数据分区。更糟糕的是，由于 Kafka 镜像的特有属性，一旦丢掉某个步骤，就根本不可能完成复制链。这时，虽然保存了大部分数据，但是并不能确定保存的是哪些数据。

Kafka 的镜像模式没有回路，还意味着不能创建多主（multi-master）系统（在这种系统下，你可以任意选择一个镜像做更新）。一般多主复制（multi-master replication）是企业环境的重要需求。MapR Streams 的镜像处理完全不同，对比学习是有帮助的。

小结

Kafka是流架构的早期革命性开创者。Kafka满足了高吞吐量以及单数据中心消息传递等很多需求，支持微服务架构，其0.9版API易于使用。不过，Kafka仍然需要大量精力手动管理存储空间和分布式。

鉴于这种设计带来的好处，Kafka及Kafka式方法引起了广泛的关注。对于多数据中心部署，Kafka还有很多严重的问题。记住，Kafka 0.9 API编写的程序还可以在MapR Streams上运行。Kafka API的灵活性或许可以为地理分布式环境下Kafka存在的问题提供一些解决办法。

第5章

MapR Streams

另一个支持流式架构的消息系统是 MapR Streams。MapR Streams 彻底重新实现了 Apache Kafka API，不仅提供 Kafka 那样的基本功能，还能提供其他功能，这将在本章讨论。MapR Streams 可以被集成到 MapR 融合数据平台（converged data platform），并且兼容 Kafka 0.9 API。Kafka 0.9 API 编写的大部分程序无须修改即可在 MapR Streams 上有效地运行。知道如何使用 Kafka，为理解如何使用 Streams 起了个好头。如果不熟悉 Apache Kafka，可以复习一下第 4 章。

MapR Streams 的创新

虽然与 Kafka 类似，但 MapR Streams 还可以做完全不同的事情。从高层视角来看，这些区别有：支持更多的主题数目，通过生存时间（time-to-live）或访问控制策略，按组管理主题。（稍后我们会介绍，这种主题组在 MapR 中被称为流（stream）。）MapR

Streams 允许设置非常多的主题数目，这样构建主题更能反映业务目标，而不受基础设施的限制。这个功能可以实现体系结构和业务问题之间的完美吻合。

将 MapR 消息系统集成到 MapR 融合数据平台，与单个集群上的消息传递技术相比，前者所需的管理性工作更少。此外，集成也简化了端到端应用程序的编写，允许流、文件和数据库在同样的安全系统下运行。

MapR Streams 的另一个重要创新是地理分布式复制（geo-distributed replication）。即使相隔很远的多个数据中心，也可以共享流数据。这种复制在任何一个数据中心更新主题，所有数据中心都能看到效果。基于消息系统的地理分布性功能强大，能够将流式设计扩展至很多有趣的使用场景，比如第 7 章的例子。

下面详细解释 MapR Streams 的不同之处：

1. MapR Streams 有一个新的文件系统对象类型，即流（stream）。在 Kafka 中，流不能并行。除了文件、目录、链接和 NoSQL 表以外，流也是 MapR 文件系统的一级对象。

2. Kafka 集群包含很多管理消息主题的服务器进程，即代理（broker）。而 MapR 集群没有与之等价的代理。

3. 在 MapR 集群中，主题和分区被存储在流对象中。而 Kafka 集群没有等价的流，只有主题和分区这样的对象。

4. 每个 MapR 流包含成百上千甚至更多的主题和分区，每个 MapR 集群可以有数百万个流。与此不同的是，任何一个 Kafka 代理上的分区数目超过 1000 时，都不是最佳实践。

5. MapR 流通过间歇性的网络连接在不同集群之间完成复制。复制模式即便有回路也没问题，而且流可以在多个不同的位置一次性完成更新。在所有复制的副本上都保存了消息偏移量。

6. 在 MapR 集群中，主题分区的分布以及各分区组成部分的分布，都是完全自动的，不需要任何管理性操作。这与 Kafka 不同，Kafka 假设很多时候管理员需要手动地重新调整分区副本的位置。

7. MapR 集群中的流继承了 MapR 基础平台所有的安全性、权限和灾难恢复能力。

8. Kafka 中的大部分生产者和消费者配置参数，都不再被 MapR Streams 支持。

在考虑大型系统的体系结构时，上述不同点非常重要。但其实我们在编写 MapR Streams 程序的时候，除了支持的主题数目更多了，很少留意到这些区别。生产者和消费者的配置有些不同，但是大部分重要参数的含义都差不多，而且很多应用程序总是使用默认值。

如果想了解更多关于为什么以及如何使用 MapR Streams 的历史，

请继续看下一小节。否则，如果只是想要知道 MapR Streams 的工作原理以及如何使用它，请直接跳到“MapR Streams 的工作原理”小节。

MapR 流系统的历史和情境

过去几年，采用 Apache Kafka 构建大型应用是令人吃惊的，它开创了一种支持流技术的方式。很多到现在还在使用 Kafka 的人都是早期使用者，这是开源项目生命周期的典型阶段。Kafka 的早期使用者通常都能很快获得惊人的成功，因为他们之前已经使用过其他 Apache 项目，如 Hadoop、Hive、Drill、Solr/Lucene 等。这些项目在各自的领域都是开创性的。在新兴技术产业实现创新的自然进化过程是，大型企业将其作为核心技术使用之后，才逐渐成熟。

为了提升这些项目的成熟度，我们需要解决可管理性、复杂性、扩展性、与其他主要技术的集成以及安全性等基本问题。过去，其他开源项目在处理这些“企业级”问题的时候，成功率很高。例如，Solr 在处理安全性问题的时候，利用边界安全（perimeter security）排除安全顾虑。Hive 社区采用 Apache Calcite 作为查询解析器和计划器，解决与 SQL 通用工具的集成问题。

有时候，在现有的开源实现环境下，这些问题很难解决。例如，Hadoop 默认文件系统 HDFS，仅支持附加文件（append-only files），不支持数量过多的文件，因为会变得不稳定，而且要求专

门的机器来维护元数据，开销相当大。想要通过逐步改进的方式真正解决已有体系结构和代码库的这些问题，是极其困难的。引入命名空间能够改善一些，但这些改变只能解决一个问题（即文件数目的有限性），甚至可能加剧其他问题（额外开销和不稳定性）。

如果在简单 API 早期，项目制定了坚实的接口标准，完全可以通过重新实现的方式解决上述问题，可能是开源实现，不过也不一定。Accumulo 是 Apache HBase 的重新实现，提供了 HBase 难以具备的功能。Hypertable 是另一个商业实现。MapR-DB 是 HBase 的第三个重新实现，类似原始的 HBase API，提供文档式的 JSON API。

同样地，HDFS 也有多个实现。有些实现使用已有的存储系统，比如 Amazon Web Services 的 S3 系统。其他实现增加了重要功能，例如，Kosmix 文件系统（KFS）向文件增加了可变性（mutability）；再比如 MapR-FS，增加了完全可变性和快照，完全去掉了名称节点（name node），性能更高，而且消除了名称节点带来的问题。有些重新实现，像比较有名的 Hypertable 和 KFS，使用范围不广，已经基本上消失了。其他像 HDFS 的 S3 接口、MapR-DB 和 MapR-FS，已经被广泛采纳，使用场景与最开始孵化 HDFS 的环境完全不同。Netflix 大量使用 S3，因为 Netflix 大量使用 Amazon 云基础设施，而很多金融、安全和远程通信应用则更喜欢用 MapR-FS，以满足稳定性和可持续性的需求。一般来说，重新实现的效果如何取决于新项目是否能够真正解决原项目难以解

决的重要问题，以及原项目的API定义是否足够清晰保证可靠实现。

Kafka存在一些比较重要的问题，其中很多问题似乎很难在原项目中解决。包括管理的复杂性、扩展性、安全性，如何处理多种数据存储模型（如文件和表），以及单一融合架构下的数据流处理问题等。

MapR Streams是Kafka的重新实现，旨在解决上述问题，同时尽量保留Kafka API的功能。虽然接口相似，但实现差别很大。本质上说，MapR Streams基于MapR技术内核，这也是MapR-FS和MapR-DB的基础。因此，MapR Streams能够重用MapR技术内核已有的很多解决方案，用于管理、安全、灾害保护和扩展等。有趣的是，只有Kafka 0.9 API才可能使用不同技术重新实现Kafka。以前的API暴露了大量内部实现细节，几乎不可能通过实质性改进的方式重新实现。

MapR Streams的工作原理

要理解MapR Streams的内部工作原理，首先要知道它与Kafka的操作方式完全不同。Kafka的内部机制在单个文件与主题分区的副本之间创建了强身份关联。发送至某个主题的消息被写入（append）分区的当前文件。如果当前文件太大，则打开一个新文件。因此，生产者发送的消息有效地顺序写入Kafka代理上的磁盘。当消费者从某个偏移量位置开始读消息的时候，只需要打开文件，

顺序读取消息并发给消费者。这些过程即快又简单。但是，这种简单性同样也给 Kafka 带来了很多限制。例如，这直接限制了每个代理上的分区数目。同样，为了确保一个主题分区的消息请求都来自一个代理（不管什么消息），一定要让这个分区完全存储在一个代理的一个文件系统中。Kafka 顺序文件 I/O 策略的关键价值在于，仅通过一个文件上相对较多的顺序 I/O 操作，就可以实现批量消息读写。

但是 MapR Streams 的基本技术很不一样。Streams 基于 MapR 数据平台的内部机制实现，例如事务（transaction）、容器（container）和 B 树等，而不直接使用通用文件功能。这些机制允许 MapR 流直接使用数据平台上的流对象，对 Kafka API 的必要功能进行建模，例如消息和生产者 / 消费者偏移量。MapR 数据平台使用分割（segmentation）技术将大型对象分布到多个容器中，并且在一个容器内复制块（block），这样实现流不会产生问题。而且，MapR 平台能够有效地将更新表式数据结构转换成大量顺序 I/O 操作，这也有助于流的实现。因此，MapR Streams 简单地从平台内核继承了这些有效的 I/O 模式，而 Kafka 则是直接实现这些转换。如果只有简单文件这一种技术核心，那么 Kafka 的策略更好；而 MapR 的策略能够快速地实现很多功能，不过前提是要有内核平台。

由于 MapR Streams 中的每个消息都要单独寻址，流的成本大部分取决于流中包含的消息总数。消息属于一个主题还是多个主题，基本上没什么区别。有一个例外是，当前正在写消息的分区数目

与并行程度稍微有点关系。这种关系比较弱，因为流的自动分割会导致分区的热区域在机器之间快速迁移，必要时在多个机器之间摊销总负载。这样，即使流中主题和分区的数目非常多，代价也不会很高昂。流中主题组——即使是非常多的主题——的管理如图 5-1 所示。例如，监测大量汽车的传感器数据，每个汽车一个主题，这个汽车所有的度量值都发送至这个主题。与之类似，网站的一个访问者对应一个主题。灵活设定哪些数据到哪个主题，以及有多少个主题，这样就可以在来数据的时候有效地将它们分到相应的会话或设备。对某些类型的分析来说，这非常有用。

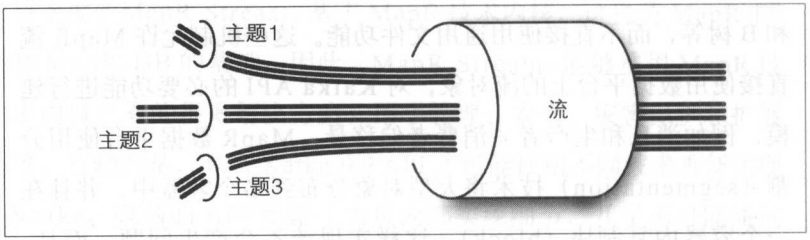


图5-1 在MapR Streams中，将主题按组分成流（stream）这样的管理结构。一个MapR流可以包含非常多的主题——多至上百万个。将主题放在一起，便于在整个主题组上应用生存时间（time-to-live）这样的策略。MapR流中的每个主题都可以被分区，与Kafka主题分区类似。（对比参考第4章图4-2）。本图中，黑色粗线表示分区。

配置 MapR Streams

MapR Streams 与 Kafka 0.9 API 的一个重要区别在于，MapR Streams 的配置属性非常少，而且有些配置是 MapR Streams 特有的。表 5-1 给出了重要的生产者配置属性，并突出了 Kafka 和

MapR Streams 共有的属性，以及生产者和消费者配置共有的序列化属性。

表5-1 MapR Streams中重要的生产者配置属性。其中，一个星号表示与Kafka共有的属性，两个星号表示生产者和消费者共有的属性。

属性	描述	默认值
buffer.memory*	生产者缓存的大小	33554432
client.id*	生产者可以对数据加标记，让消费者知道源是谁	None
key.serializer**	生产者和消费者键的序列化程序	None
value.serializer**	生产者和消费者值的序列化程序	
streams.buffer.max.time.ms	生产者在发送数据之前的缓冲时间	3000ms
streams.producer.default.stream	不以“/”开头的主题对应的主题名称	

这些差异是配置生产者流的关键。由于 Kafka 没有相应的流概念，MapR Streams 采用主题名称定义当前正在使用的流。命名规则是流的全路径，后面是斜杠或冒号，再加上主题名称。如果设置了 streams.producer.default.stream 属性，对所有不以斜杠开头的主题名称来说，这个属性的值就是主题名称。注意，没有 bootstrap.servers 属性去帮助生产者或消费者连接代理集群，MapR Streams 程序不需要这个，因为没有代理要连接。

表 5-2 给出了消费者的配置属性。同样，星号表示 Kafka 和 MapR Streams 共有的属性，同样由于没有代理，因此没有 bootstrap.servers 属性。

在消费者端，有趣的是 streams.consumer.buffer.memory 属性，设置消费者从流中预读取（pre-fetch）的数据量。在 Kafka 中，只有 KafkaConsumer 对象调用 poll() 方法，才会从代理读数据。而 MapR Streams 从订阅的主题中预读取数据，允许读取和计算交叉进行。这样，消息读取基本是零延迟。不过，预先读取并不影响端到端的延迟（end-to-end latency）。

表5-2 MapR Streams中重要的消费者配置属性。星号表示MapR Streams与Apache Kafka共有的属性。

属性	描述	默认值
auto.commit.inerval.ms*	如果 auto.offset.reset 为 true，多长时间提交一次偏移量	
auto.offset.reset*	确定消费者组新的偏移量位置，值为 earliest、latest 或 none	latest
enable.auto.commit*	启用自动提交主题偏移量	true
fetch.min.bytes*	如果服务器上可用的字节数比这个值少，则阻塞请求，直到可用字节数达到这个数目	1 字节
fetch.max.wait.ms*	字节数不够返回的情况下，请求需要等待的时间	
group.id*	这个消费者对应的消费者组名称	
max.partition.fetch.bytes*	在每次请求中，消费者试图从流中读取的数据量	64kB
streams.consumer.buffer.memory	指定预读取消息所需的内存大小	64MB
streams.consumer.default.stream	指定不以“/”开头的主题的默认流	

在生产者 API 中，可以使用简单的主题名称设置默认流。

地理分布式复制

MapR Streams 支持多项数据中心到数据中心的复制功能。这种复制比 Apache Kafka 的 MirrorMaker 函数更强大。关键是使用类似 MapR-DB 的近实时复制技术。这种近实时复制能够连续传输变更记录。有趣的是，复制图（replication graph）可以包含回路而不会出错，因为可以检测到那些通过多条路径传输的记录。

复制模式如图 5-2 所示的。案例 A 简单展示了两个数据中心之间双向复制：向其中一个数据中心的复制流（replicated stream）插入消息，然后尽快将变更复制到另一个数据中心。案例 B 是一个更为复杂的例子，在旧金山和新加坡的数据中心之间，以及新加坡和悉尼的数据中心之间双向共享数据。可以向任意一个数据中心的复制流插入消息，但是给定主题一次只能在一个数据中心插入消息。案例 C 中没有复制回路。数据从纽约复制到伦敦，以及从巴黎复制到伦敦，但是不会往回复制。设置生存时间，使纽约和巴黎的数据只保留较短的时间，而伦敦的数据可以保留很长时间。这种复制数据的能力对数据获取的场景非常有帮助。对于以上三个案例，Kafka 只能实现最后一个。

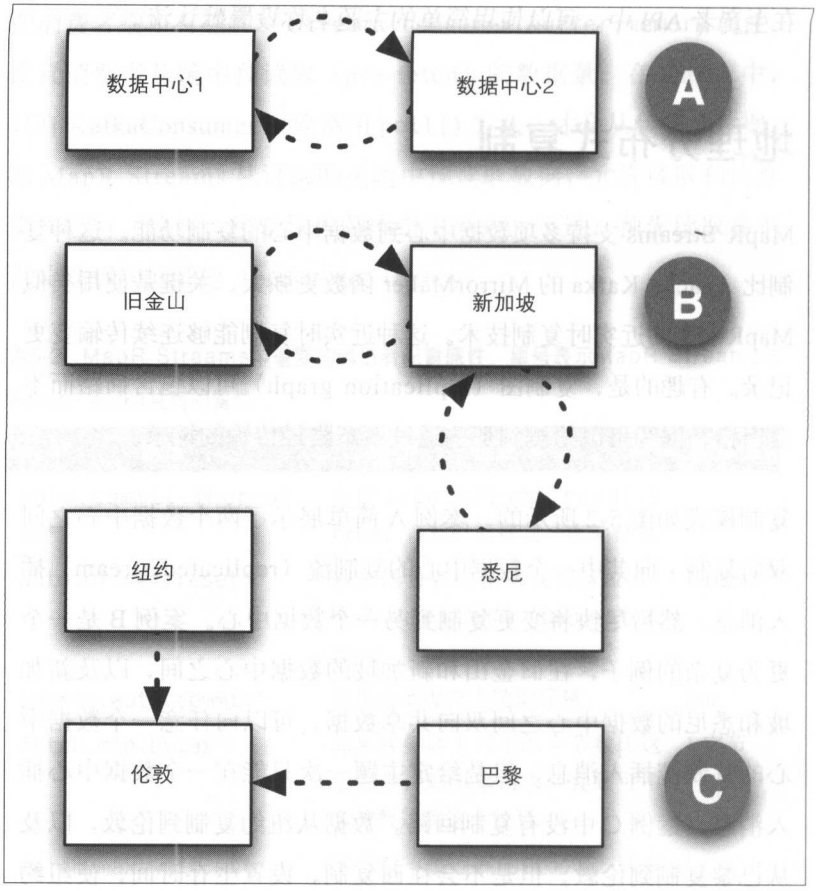


图5-2 在MapR Streams中，流在集群之间复制，而且允许循环复制。这在Kafka MirrorMaker中是不可能的。

MapR Streams 的陷阱

MapR Streams 是否适合你的流架构项目，有很多影响因素。2016 年年初，MapR Streams 发布了第一个 GA 版本[注1]，相对来说还是比较新的项目。但它并不像看上去那样不成熟，因为早在 2010 年就已经开始测试底层技术，至少目前来看，MapR Streams 绝对是值得考虑的选择。Kafka 0.9 API 也是新的，两者都是既年轻又成熟的系统。

有个问题是，MapR Streams 只支持 Kafka 0.9 API，不直接兼容基于 Kafka 0.8 或更早版本的 API 构建的应用。这些应用想要使用 MapR Streams，还需要很多额外的工作。那些想要升级到 Kafka 0.9 的 Kafka 用户，也要做出同样的努力——不得不重写（修改）原来的 Kafka 应用，以便在新 API 上运行。

这两个系统之间的还有一个有趣的区别，Kafka 的管理员角色更加活跃，有时候还必须明确集群中主题分区的位置。而在 MapR 集群，用户对于存储和访问主题分区的控制要少得多。流可以通过卷拓扑（volume topology）被限定在大型集群的某一部分，但是不能提供 Kafka 那样的手动数据定位功能。这是优点还是缺点呢？具体要取决于你的应用，以及在你的环境下低层次的管理是好还是坏。

使用 MapR Streams 的地理分布式复制功能，应该注意几个问题。首先，虽然整个流都能进行多主复制（multi-master replication），

注 1　General Availability，正式发布的版本。——译者注

但是复制流中的一个主题应当只向一个副本写消息。如果所有副本都是最新的，可以设置哪个副本最先得到消息，但是不要将消息同时写入不同地理位置上相同主题的多个副本。通常做法是，某个位置上的主题负责接收来自这个位置的数据，或者更改复制速率相对较慢的主题对应的插入点（insert point）。

MapR Streams 程序示例

可以在 *http://bit.ly/mapr-streams-code-samples* 上查看如何编写 MapR Streams 代码的详细示例，以及 GitHub 应用实例的代码库链接。记住，MapR Streams 编程是基于 Kafka 0.9 API 的。

第6章

基于流数据的欺诈检测

信用卡欺诈检测是流数据解决方案在金融领域的重要应用之一。这一章我们讨论流式架构怎样为信用卡欺诈检测提供更好的架构基础。而且如果设计巧妙，还能得到除欺诈检测这一首要目标之外的其他好处。

因此，我们有两个主要设计目标：

目标 1

> 当客户进行信用卡交易的时候，商家需要迅速知道“这是不是欺诈？”

目标 2

> 保存系统的欺诈决策历史。组织内的其他应用和服务，以及欺诈系统内部的数据库更新，都可以使用这个决策历史。

刷卡速度

在这个信用卡例子中，为了强调架构，我们将欺诈检测工作极简化。我们采用属性刷卡速度（card velocity）表示欺诈活动的可能性。刷卡速度的思想很简单：假设信用卡在伦敦的某个 POS 机使用，三分钟后，同一张卡又在澳大利亚的悉尼被刷。能得到什么结论？除非有人发明了分子传输器或时间机器，否则，其中一次信用卡交易就是欺诈，或者两次交易都是欺诈。

识别欺诈活动听上去很容易，实际上也不难，但是需要针对上百万个交易非常迅速地做出决策。首先要确定每个交易是欺诈的相对可能性，然后必须以可靠的方式将结果返回 POS 机，延迟只能有几十毫秒。这就是基于机器的决策，消息流处理系统非常重要。图 6-1 展示了这个示例系统的高层结构。决策引擎用于确定交易是不是欺诈，即图中带问号的方框，稍后我们会解释方框里面的内容。注意，多个 POS 终端可以向同一个决策引擎发送请求并得到返回决策。

请注意，为了叙述方便我们简化了这个例子。实际欺诈检测系统确实将刷卡速度作为判断信用卡欺诈的一个指标，但这只是上百个指标中的一个。这些系统很复杂，包括很多不同权重的指标。这里我们强调整体架构，因此尽量保持简单。

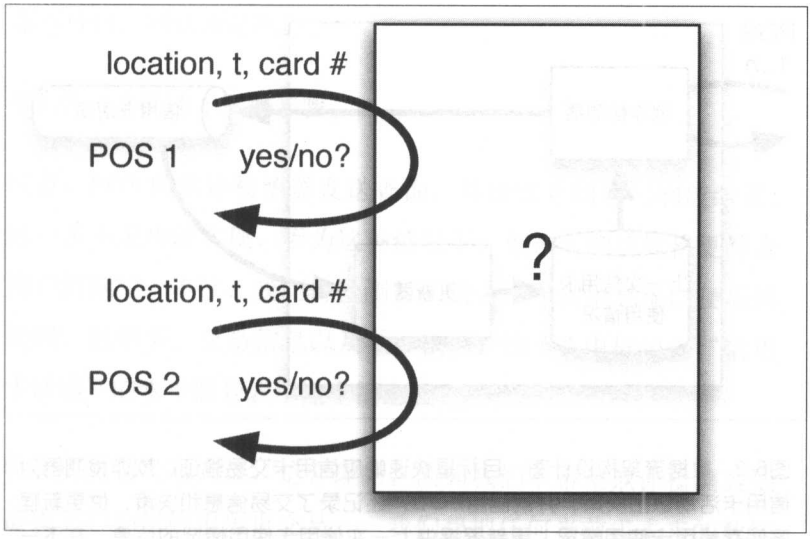


图6-1 当发生信用卡交易时，POS机向数据中心发出请求，询问本次交易是否可能是欺诈（是／否？）。本次事件请求包含交易的时间、地点及卡号等数据。几乎同时产生大量的信用卡交易请求，约50~100毫秒内就需要做出决策并给出答案。

快速响应决策："这是欺诈吗"

我们基于刷卡速度检测可能是欺诈的异常交易。每个交易都包含地点、时间，以及相关联的信用卡信息，如卡号。欺诈检测应用的每一次请求都要传送这些信息，如图 6-2 所示。

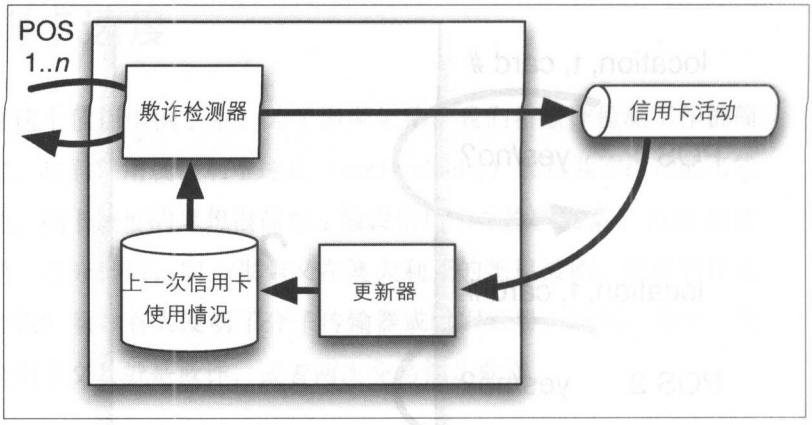


图6-2 数据流架构设计图，目标是快速响应信用卡交易验证。欺诈检测器对信用卡活动做出决策，并返回给POS；还记录了交易信息和决策，供更新程序保存信用卡使用情况。更新器提供上一次信用卡使用情况的信息，在下一次做欺诈决策时作为参考数据。

这里不深入讨论欺诈检测应用的细节（我们已经在其他出版物中讨论过类似的内容），而会从高层抽象的角度来看看训练系统学习正常行为所涉及的方面，包括连续两次交易的时间间隔，以及两个位置之间的距离。在这个简化的例子中，模型只需要保存上一个地址即可。然而，更复杂的模型需要收集信用卡交易历史，训练更复杂的检测模型。尽管目前不太需要，我们还是来设计一个便于收集交易历史的架构。

为了做出欺诈决策，需要检查信用卡以前的使用情况，从中抽取出位置和时间信息，计算当前交易的刷卡速度。如果刷卡速度太

高不可信，则认为是欺诈。

架构图 6-2 描述了这些步骤。

注意，POS 向欺诈检测器发送查询，并希望立刻返回响应结果。这一步不采用流实现，因为这种情况下，查询 - 响应风格更符合用户的期望。不过，我们将检测器的结果发布到消息流以便后续处理，包括卡、交易信息以及欺诈决策。图 6-2 中标记为“信用卡活动”的水平圆管表示这个消息流。

将历史交易发布到流，是合理的，因为我们不希望欺诈检测器知道或关心如何使用历史交易——我们想把任何当前或未来的数据消费者从欺诈检测器完全解耦出来。交易信息流确实有助于构建更为复杂的欺诈检测机制，可付诸实际应用。

另一方面，即使这个简单的例子也需要数据库来存储上一次交易信息，以便按照卡号查找交易。为了构建这一机制，每次做决策的时候，欺诈检测器会直接更新数据库。在实践过程中，更好的方式是将所有交易都发送到欺诈检测器之外的消息流，然后再将这些信息导入欺诈检测器内的数据库。这个设计有点绕，但是如果需要扩展这个欺诈检测原型，这种设计的合理性就会突显出来。另外，数据库可以采用 Apache HBase 或 MapR 集成的 NoSQL 文档式数据库，即 MapR JSON DB。

图 6-1 所示的数据流满足了第一个目标，图 6-2 所示的消息流（Kafka 或 MapR Streams）还有助于配置其他项目，即满足了第二个目标。

多用途流数据

欺诈检测器原型系统构建得差不多了，现在加入更多需求。例如，无论谁负责构建下一代欺诈检测模型，他都会想要分析历史数据。除了欺诈检测以外，还有别的项目可能要用到信用卡交易数据。

流式架构系统很容易添加其他服务。基于刷卡速度的欺诈检测数据流设计，使信用卡活动数据用途更加广泛，如图 6-3 所示。将信用卡活动消息队列暴露给其他内部服务，其他服务就能够看到完整的授权过程。记住，这个数据流中包含欺诈检测器输出的决策结果。

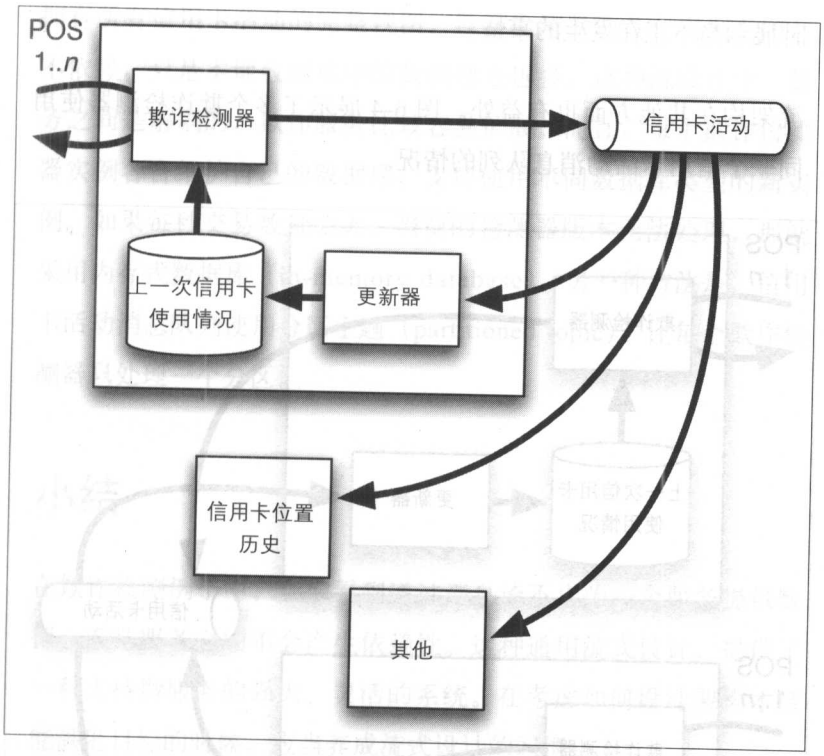


图6-3 使用消息队列作为欺诈检测器输出的好处。信用卡活动和欺诈决策数据的用途更广泛。队列中数据的存留时间更长，也很有帮助。例如，进程可以查询数据，收集信用卡位置历史，以Parquet这样的数据格式存储，保证非常高效的查询。其他进程可以使用信用卡活动队列中的数据，确定订阅者中的欺诈比率，或实时展示交易发生时的情况。

欺诈检测器的向上扩展

我们刚刚学习了使用流架构构建欺诈检测器原型，它也允许其他服务进行访问数据，以便通过历史性分析改进决策模型，或者实

时展示当下正在发生的事情。

流架构在其他方面也有益处。图 6-4 展示了多个欺诈检测器使用同一个信用卡活动消息队列的情况。

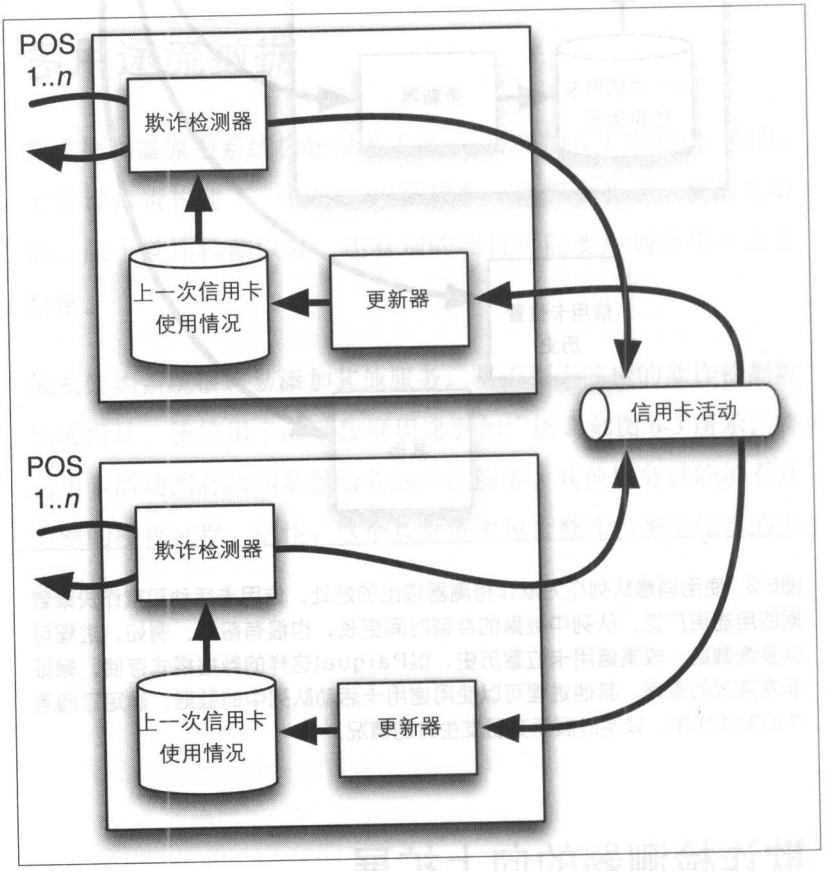


图6-4 多个欺诈检测器可以使用同一个消息队列。每个欺诈检测器使用所有的信用卡活动数据，因此能够处理任何交易。

所有欺诈模型将全部交易发送到信用卡活动消息队列，因此队列

拥有所有信用卡活动的完整视图。欺诈检测器的各个实例之间互不依赖，只是本地数据库中的数据越来越多。这种流设计中，服务之间是解耦的，欺诈服务比较容易扩展。而且，每个欺诈检测器实例各自维护自己的数据库，支持使用不同数据库类型的新实例。如果每秒交易数量太大，当前的检测器版本无法处理，那就采用内存式数据库（in-memory database）。另一种方法是，信用卡活动消息队列使用分区主题（partitioned topic），让每个欺诈检测器只处理一个分区。

小结

在欺诈检测例子中，我们看到通过消息流不止为一个服务提供数据，而且服务之间不会产生依赖性。这种通用流式设计，提供了一种支持微服务的强大、灵活的系统。在考虑如何设计架构才最能满足目标的时候，应当养成流式设计的习惯。

第 7 章

地理分布式数据流

来章来看最后一个关于设计流式系统的例子，我们考虑一种特定的需求：数据流的地理分布式复制。大量应用场景都需要这个功能，包括远程通信、油气勘测、零售业、银行业等。我们以交通运输——国际集装箱运输——为例展示系统的数据流动，该系统要求在不同地理位置上有效地进行数据复制。

我们以 MapR Streams 为例来讨论如何设计，因为它特有的功能非常适合这种使用场景。MapR Streams 独特的功能包括：

- 处理超大数量的主题（成百上千甚至更多的主题，而且吞吐量很高）。
- 以组的形式管理主题，即流，主题组使数据管理更加简单。
- 在地理上分散的数据中心之间提供简单可靠的单向和双向复制。

在这个运输（或任何其他领域）的例子中，MapR 的消息传递功

能由集成的数据平台提供，同一集群内存在很多不同的进程和消息传递。但是，为了使这个例子简单地体现数据流在不同站点之间的复制，我们只关注架构中消息传递的部分，忽略分析和持久化的部分。

当我们在讨论物联网运输案例的时候，想象一下这种设计对你自己的项目有什么作用。你可能没有船只和集装箱，但是数据需求可能是相似的。

利益相关者

假设这样的场景：在国际集装箱运输中，不同的利益相关者以不同的方式使用运输数据。如何根据每个利益相关者的需求管理数据？每个主题需要分配什么样的数据？有多少个主题？哪些主题可以分组成流？什么时候需要使用地理分布式流复制？

就这个例子来说，没有哪一个设计是对的，但是通过研究这些设计方案，就可以在自己的场景下更好地决定如何在主题和流之间分配数据。

我们虚构一个名叫 Big Blue 的大型运输公司，它拥有一个船队，以及成千上万个集装箱。Big Blue 总部在洛杉矶，向全国各地运输货物，包括东京、悉尼、新加坡，如图 7-1 所示。Big Blue 不仅用自己的集装箱运输，还可以把空间租售给其他运输公司或大型厂商。Big Blue 需要知道船在哪里、哪些集装箱到了哪些港口，

以及船上和岸上的环境条件怎么样。Big Blue 在很多主要港口都设有办事处，他们希望这些港口的数据在各港口之间实现共享，并向总部汇报。

图7-1 夜晚的新加坡。全球的集装箱运输很大部分都要经过这个港口，大量集装箱的装箱和卸货数据都通过传感器监控。

还有谁对运输数据感兴趣呢？另一类利益相关者是向 Big Blue 货物运输买单的厂商。在公司总部，厂商要知道货物的状态，比如运输过程中的环境条件，以及这些货物的目的地。购买货物的收件人也想知道这些信息，不过粒度可能没有那么细。每个港口的港务局也有自己感兴趣的数据。他们不仅关心 Big Blue 的船和集装箱，还要确切地知道哪个公司的哪艘船靠岸或离开，或者卸下

了什么货。以上这些都是主要的利益相关者。

设计目标

我们已经知道了主要的利益相关者有哪些，现在我们简单总结一下设计目标。重申一下，我们把例子极简化，重点强调数据的地理分布式特性，以及消息传递层的数据流动。我们的设计必须实现以下几点：

- 从连续的事件和度量值（包括物联网传感器数据）中可靠地收集、分析和保存超大规模的数据，保证高性能和低延迟。
- 解耦源（如传感器）的数据交付与数据中心的整合和处理。
- 通过管理数据的可见性进行访问控制。
- 当船在海上的时候，解决船到岸的间歇性数据传输问题。
- 在地理分布式集群之间（包括港口到港口、港口到厂商或总部，以及船到岸）有效地复制数据流。

设计选择

以上述目标为指导，我们开始设计方案。以某次从东京运输塑料玩具鸭为例。该厂家的总部在东京，Big Blue 运输玩具鸭的过程如图 7-2 所示。船到达东京（A），装载玩具鸭等货物，然后去往新加坡（B）。在新加坡卸载部分玩具鸭集装箱，其中一些送到当地的奥特莱斯，剩下的继续转运到伦敦（图中没有显示）。其余

的玩具鸭集装箱，以及在新加坡新装载的其他公司旗下的集装箱，则继续运往悉尼港口（C）。

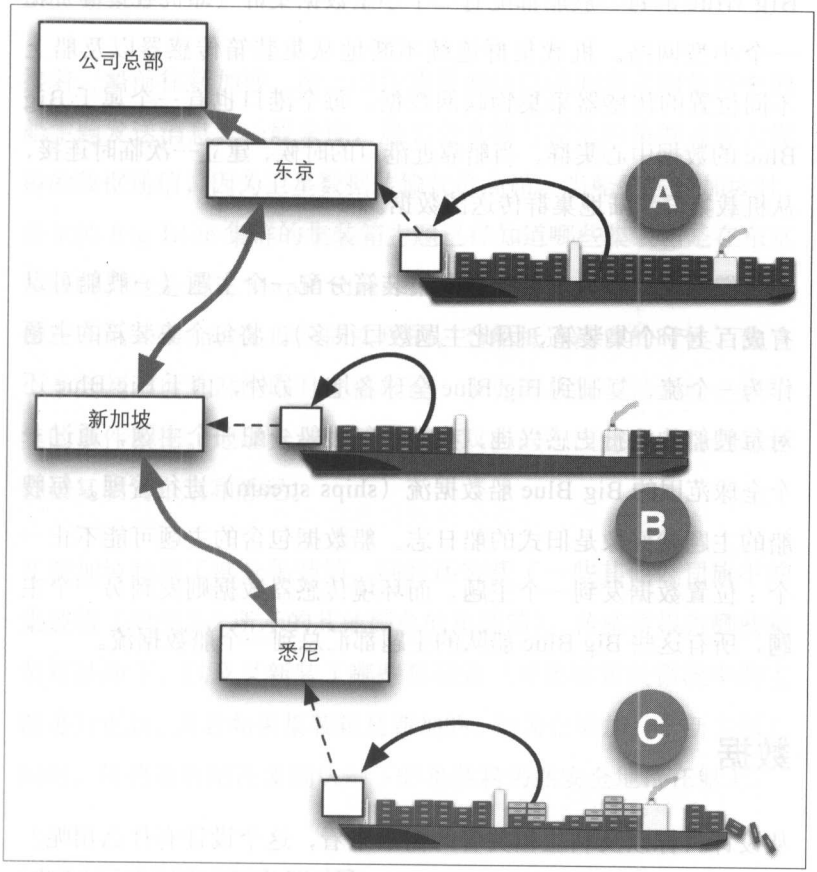


图7-2　集装箱运输公司的数据流动。环境传感器和集装箱跟踪传感器，连续不断地向运输公司的机载集群（白色方框）传送数据流（黑色箭头）。当船到达港口的时候，从机载集群到运输公司的陆地数据中心集群（未显示）之间形成流数据的临时连接（虚线箭头）。同时，在不同港口的数据中心之间双向复制流（双向灰色箭头）。

我们的设计

Big Blue 的每一艘船都配有一个小型数据集群（即机载集群）和一个小型网络。机载集群连续不断地从集装箱传感器以及船上不同位置的传感器采集物联网数据。每个港口也有一个属于 Big Blue 的数据中心集群。当船靠近港口的时候，建立一次临时连接，从机载集群到陆地集群传送流数据。

在我们的设计中，我们给每个集装箱分配一个主题（一艘船可以有成百上千个集装箱，因此主题数目很多）。将每个集装箱的主题作为一个流，复制到 Big Blue 全球各地。另外，由于 Big Blue 还对每艘船的旅行史感兴趣，我们给每艘船分配一个主题，通过一个全球范围的 Big Blue 船数据流（ships stream）进行管理。每艘船的主题就好像是旧式的船日志。船数据包含的主题可能不止一个：位置数据发到一个主题，而环境传感器数据则发到另一个主题，所有这些 Big Blue 船队的主题都汇总到一个船数据流。

数据

从设计目标以及利益相关者的需求来看，这个设计有什么用呢？参考图 7-2，我们看一下这个系统。从步骤 A 开始，船在东京装载，该船的机载集群向东京 Big Blue 集群提供有关该船的数据更新（船数据流中的两个主题），以及船上装载了哪些集装箱，哪些集装箱装了玩具鸭、哪些装了其他货物（一个集装箱一个主题，向集装箱数据流中的上千个主题发送更新）。东京 Big Blue 集群向玩具

厂商公司总部（即图 7-2 中的“公司总部”）汇报这些信息的子集。同时，这些更新还会实时传播到其他港口的数据中心，以及 Big Blue 总部。

接着，船前往新加坡，船上的传感器继续以流的形式向集群中的船主题发送消息。一般来说，船不会直接与岸上的集群进行大规模的数据通信，因为卫星数据传输代价高昂。当船到达新加坡时，新加坡 Big Blue 集群的集装箱主题已经知道哪些集装箱是在东京装载的。这是通过 MapR Streams 的地理分布式流复制功能，直接在东京和新加坡集群之间实现的。当船抵达港口的时候，建立一个到新加坡集群的临时连接，进一步更新从集装箱到船之间收集的事件数据。地理分布式流复制是双向的，因此这些新信息还会被复制到东京和悉尼。

在新加坡卸载了部分集装箱，同时还装载了一些其他公司旗下的集装箱（即图 7-2 所示的其他颜色的集装箱）。传感器报告哪些集装箱被卸下，以及又新装了哪些集装箱（对集装箱数据流中的主题进行更新，或者如果集装箱是新加的，则为它增加一个新主题）。同时，传感器数据还要确认剩下的集装箱仍然安全地待在船上。

控制谁能访问流数据

我们还要满足另一个设计目标。场景是：新集装箱的拥有者想要访问这些集装箱有关的消息数据，但是 Big Blue 不想让他们访问全部数据。幸好，MapR Streams 可以控制谁能访问数据。在流层设置访问控制表达式（Access Control Expression, ACE）就可以

分别为黄色和红色集装箱主题设置独立的流。这样，就能允许客户访问关于他们自己集装箱的主题数据，同时限制他们访问 Big Blue 的流数据。

继续回到船：接下来，船开往悉尼。同样，在船到达港口之前，数据就已经到达。船将数据上传至岸上的新加坡集群，通过 MapR Streams 复制到悉尼集群。发生在新加坡的主题更新触发复制。当船到达悉尼的时候，再次建立船到岸的临时连接，向悉尼集群发送廊道上的事件数据。

当船进入港口以后，集装箱上的传感器继续向岸上集群提供数据流，报告集装箱的状态，有些集装箱被卸载，有些还留在船上。当玩具鸭集装箱从船上滑落时（图 7-2 中的步骤 C），触发警告信息，并被发到机载集群，这是从传感器接收到消息流。这些信息立刻被复制到港口集群，以及洛杉矶 Big Blue 总部。假如迫不得已要向玩具厂商汇报玩具鸭遗失了（检查各地的情况以确定最后看见玩具鸭的地方），管理者肯定不高兴。[注 1]

基于流的地理分布式复制的优势

在这个“玩具例子”（双关语）中，每个集装箱的主题以及主题组构成的集装箱流，为这个集装箱提供了连续的历史记录，尽管这

注 1　这个例子源自 1992 年从香港出发的货船在太平洋海域丢失玩具鸭、玩具乌龟、玩具海狸、玩具青蛙等“漂浮物”的真实事件。1992 年年底，部分漂浮物到达阿拉斯加。2007 年，大部分漂浮物在英国海岸被发现。更多信息请参考 *https://bit.ly/lost-ducks*。

个集装箱可能在多个船或多个港口码头逗留过。基于流的组织非常方便，因为可以在流层上设置生存周期、地理分布式复制和数据访问控制。

这家国际运输公司拥有大量集装箱，每个流需要处理成百上千甚至更多的主题。目前，在大规模主题之间的通信问题上，MapR Streams 的处理能力不同寻常。同时，MapR Streams 还能进行多主（multi-master）复制和地理分布式复制。我们选择这个特定的案例是因为它涉及数量的主题、间歇性网络连接、流客户端故障转移以及地理分布式，很多使用场景都存在以上问题，只是有时候不那么明显。

第8章

总结

学了前面的内容，我们能够做什么呢？

让我们重新审视一下目标，看看切换到通用流式方法以及实时分析应用能够为你带来什么好处。

事实上，很多应用场景下使用流数据都能带来革命性的变化，包括物联网传感器数据行业、金融服务、远程通信、互联网商务、零售业、医疗健康等。高速、大规模连续性事件数据处理方面的新技术是促成这场变革的原因之一。另一个要素就是基于这些新兴技术的新型架构设计方法。通用流式设计能够带来很大变化。这并不是说流数据可以做所有事情，而是意味着流已经成为一种通用方法，不再只是适用于某些特殊的实时项目。

当大数据架构采用流式设计成为一种习惯时，将取得巨大收益。

流式架构的核心是消息传递。流设计与传统设计（包括人们以前先入为主的流概念）的主要区别在于，消息传递层的重要性更加突出。虽然流数据处理是这些应用的核心，但是消息传递能够而且应该不止是实时分析前的一个步骤。有效的消息传递需要 Kafka 这样的工具。新技术仍在持续发展，不过目前我们认为 Apache Kafka 和 MapR Streams 是流式系统消息传递层的最佳选择。不论选择哪一种消息传递技术，都应当考虑是否具备以下重要功能。

消息传递技术的关键要素

要想真正得到流架构的好处，消息传递技术需要具备以下特质：

- 可重放（replayable）
- 可持久化（persistent）
- 大规模下的高性能

对于现代流架构的处理组件，有很多技术可供选择。例如 Apache Spark Streaming 和 Apache Flink 项目，二者在技术上有些不同。Spark Streaming 是 Spark 的一个附加功能，利用内存式计算的速度优势解决一种特殊的批处理——微批处理（microbatches），实现近实时分析。Flink 是一项新技术，不仅提供大规模下的高速计算，还能实现实时流处理，必要的时候还可以被割成批处理。以上两个系统都是构成消息传递层非常具有吸引力的选择。

流式架构的优势

基于有效消息传递的流式架构的优势之一是数据移动更少、速度更快。这种方法很便捷：所需的管理更少，移动和协调各部分的工作更少。它能够强有力地支持微服务，进而使组织更加敏捷。在体系架构设计中，合适的消息传递组件应该解耦服务；数据源不需要协调消费者。这就是为什么消息的持久化是很重要的：如果在交付消息的时候消费者不可用，没有关系——消息能够在需要的时候可用。这并不是说查询-响应方法就再也没有用武之地了，而是说基于流的消息传递层在很多场景下都非常强大。

流式架构的另一个优势在于灵活性，即以不同的方式向多个消费者提供数据。这就强调了原始数据的交付和持久化。因为在设计架构和数据流的时候，并不知道可能需要这些数据的全部应用，也不知道最终哪些数据是重要的。

有效地处理流数据，更易于响应不断变化的事件，并且通过实时的洞察力对正在发生的事情做出反应。

数据流的地理分布式复制极大地拓展了流式架构的影响。第 7 章介绍了一个案例，其中在多个数据中心之间快速共享流数据的能力在很多地方都适用。目前，MapR Streams 是最合适的消息传递工具。

过渡到流架构

在规划新项目的时候，基于流架构进行设计非常简单，而且还能为以后的修改提供更大的灵活性。但是，如果仍然有遗留的旧服务，如何融入流式架构呢？

将应用迁移成消息传递风格的实现，比你想象的要简单得多。而且，消息传递给予的灵活性，能够有效、便捷地将变更融进遗留项目。工作原理如下。

传统架构的一个问题是，即使对于最初设计的应用场景来说作业效率很高，但是一旦尝试添加服务或是进行修改，就会变更非常困难。造成这种情况的部分原因在于服务之间的强依赖性，如图8-1所示。

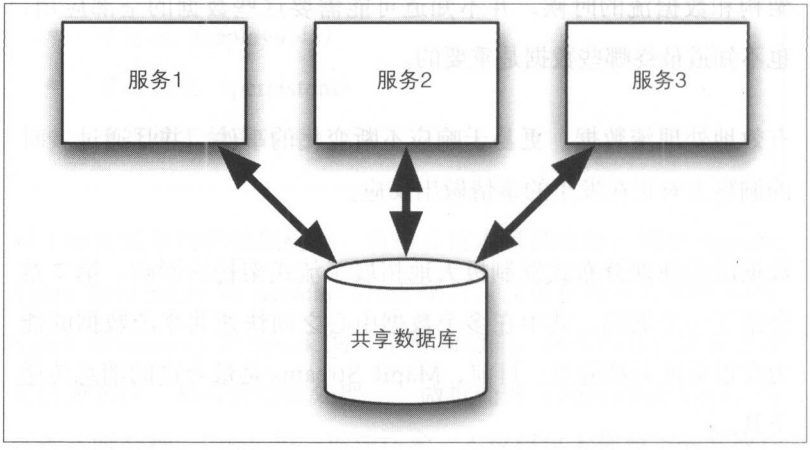


图8-1　传统架构中，组件是强耦合的。在本图中，服务1、服务2和服务3使用存储在共享数据库中的数据，并直接更新数据库。这种结构或许是有效的，但是如果想要修改任何服务，由于存在依赖性，会导致整个系统都要做不必要的变更。

假设我们想修改其中一个遗留服务。传统设计中组件的耦合性意味着服务 1 中的变更会同时影响服务 2 和服务 3。这就是说，在设计服务 1 的时候，就要考虑到其他可能受影响的服务。反之亦然，但那样就陷入死循环了。但是，如果在服务和数据库之间添加一个消息队列，变更就很容易了，如图 8-2 所示。

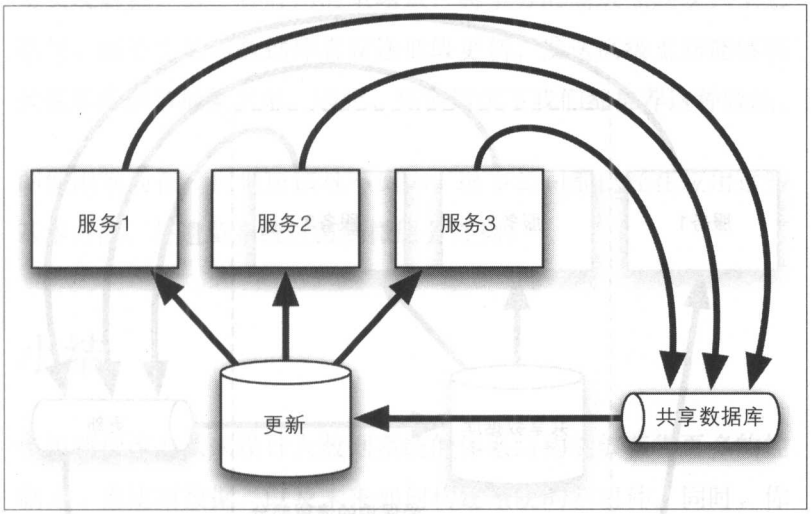


图8-2 使用流式架构方法重新设计遗留系统，在服务和共享数据库之间添加一个消息流做更新。来自服务的更新传到流，然后再到数据库。

添加消息传递层实现更新，就可以修改服务，而不会给其他服务带来不必要的影响。这里，来自服务 1、服务 2、服务 3 的全部更新在抵达共享数据库之前，都要经过消息队列这个中间步骤，即图 8-2 中名为“更新”的圆管。服务 1、服务 2、服务 3 是生产者，共享数据库是消息队列的消费者。这个中间组件解耦了数据的生产者和消费者。

下面变更服务 1。首先做一个数据库备份，这个备份数据库不共享，如图 8-3 所示。服务 1 从备份数据库读数据，而服务 2 和服务 3 继续从共享数据库读数据。注意，所有更新仍然发到同一个消息队列，只是现在多了一个不共享的数据库作为数据消费者。这样就实现了服务 1 与遗留服务之间的隔离。

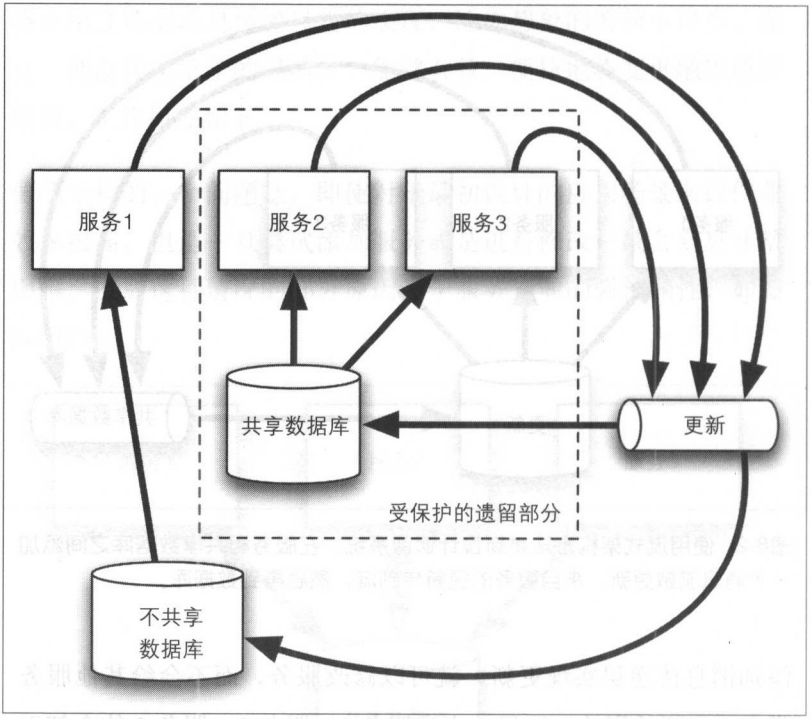


图8-3　流式架构的变更更加简单，因为组件之间是解耦的。这里，添加一个数据库备份，将服务1与遗留服务和共享数据库隔离开来。三个服务的更新都发送到同一个消息流，遗留服务只能订阅部分更新，而服务1可以订阅所有更新。这样，变更服务1带来的影响就不会影响系统的遗留部分。

这个例子非常简单，有助于我们理解在从传统系统过渡到新的流架构的过程中，解耦所起到的作用。在实际解耦的过程中可能会出现一些问题。最严重的问题之一就是数据库事务性更新（transactional update）的依赖性。解决办法是，将事务性更新隔离到单个服务内，或者服务将数据库视作纯消费者，这样就很容易实现解耦。另一种有用的策略是，将事务的高级描述发到消息队列，而不止是向数据库表发送低级更新。发送高级更新能够将数据库的细节抽象出来，因此，任何情况下我们都推荐这种做法。

即便困难尚在，但也应该难不倒你。很多公司都已经在应用这些基本方法，将遗留系统过渡到微服务架构。

小结

使用新型流方法来设计大数据系统的体系结构能够获得更多的控制力：谁使用数据，以及未来如何构建系统的新组件。同时，你也加入了流数据浪潮，并得益于此。

> “总的来说，流技术的作用很明显：连续性地处理数据，这些数据是由现实世界的数据源连续不断地自然产生的（即‘大’数据集）”。[注1]
>
> —— Fabian Hueske 和 Kostas Tzoumas，Apache Flink 的贡献者和 PMC 成员

大多数时候，流数据确实更符合真实世界的客观规律。

注 1　http://bit.ly/guide-stream-processing-flink

附录 A

附加资源

流数据主题

下面给出关于流设计及其相关技术的链接和推荐读物。

Apache Kafka

- 项目网址为 *http://kafka.apache.org/*。
- 博文 *Getting Started with Sample Programs for Kafka 0.9*，网址为 *https://mapr.com/blog/getting-started-sample-programs-apache-kafka-09/*。

MapR Streams

- 博文 *Life of a Message in MapR Streams*，网址为 *http://bit.ly/mapr-streams-message*。
- MapR Streams 说明文档，网址为 *http://bit.ly/mapr-streams-doc*。

- 博文 *Getting Started with Sample Programs for MapR Streams*，网址为 *https://mapr.com/blog/getting-started-sample-programs-mapr-streams/*。

I Heart Logs

作者是 Jay Kreps，他是 Apache Kafka 的贡献者和 PMC 成员。本书由 O'Reilly 出版社出版，网址为 *http://bit.ly/i_heart_logs*。

Apache Spark Streaming

项目网址为 *http://spark.apache.org/streaming/*。

Getting started with Apache Spark

作者是 Jim Scott，这是一本免费电子书，可从 MapR 下载，网址为 *https://mapr.com/getting-started-apache-spark/*。

Apache Flink

项目网址为 *https://flink.apache.org/*。

Essential Guide to Streaming-first Processing with Apache Flink

作者是 Fabian Hueske 和 Kostas Tzoumas，他们是 Apache Flink 的贡献者和 PMC 成员，该博文网址为 *http://bit.ly/guide-stream-processing-flink*。

Apache Storm

项目网址为 *http://storm.apache.org/*。

Apache Apex

项目网址为 *http://apex.apache.org/*。

英国谢菲尔德大学 - 波音公司先进制造技术研究中心，包括工厂 2050

网址为 *http://www.amrc.co.uk/*。

作者在 O'Reilly 出版的其他作品精选

下面简单列出几本有趣的大数据领域的书籍。

- *Practical Machine Learning: Innovations in Recommendation*，2014 年 2 月出版，网址为 *http://shop.oreilly.com/product/0636920033172.do*。
- *Practical Machine Learning: A New Look at Anomaly Detection*，2014 年 6 月出版，网址为 *http://bit.ly/anomaly_detection*。
- *Time Series Databases: New Ways to Store and Access Data*，2014 年 10 月出版，网址为 *http://shop.oreilly.com/product/0636920035435.do*。
- *Real-World Hadoop*，2015 年 3 月出版，网址为 *http://shop.*

oreilly.com/product/0636920038450.do。

- *Sharing Big Data Safely: Managing Data Security*，2015年9月出版，网址为*http://shop.oreilly.com/product/0636920045571.do*。

作者简介

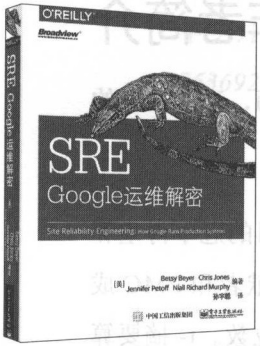
Ted Dunning，MapR Technologies 首席应用架构师，开源社区的活跃成员。

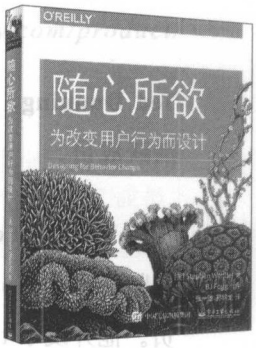
现任 Apache Foundation 孵化器的 VP，是大量项目的冠军得主和导师，也是 Apache ZooKeeper 和 Drill 项目的贡献者和 PMC 成员。他开发了 t- 摘要（t-digest）算法估算极端分位数。t- 摘要算法已经被几个开源项目使用。他还开发了开源项目 log-synth，在 *Sharing Big Data Safely*（O'Reilly 出版）一书中有所介绍。

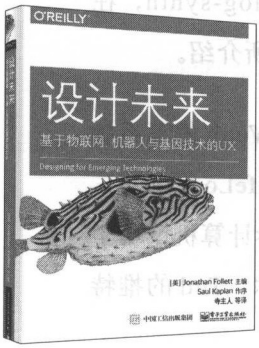
Ted 曾经是 MusicMatch（即现在的雅虎音乐）和 Veoh 推荐系统的首席架构师，创建了欺诈检测系统 ID Analytics（LifeLock 公司），至今共获得 24 项专利。Ted 拥有英国谢菲尔德大学计算机博士学位。除了数据科学以外，他还喜欢弹吉他和曼陀林。Ted 的推特账号是 @ted_dunning。

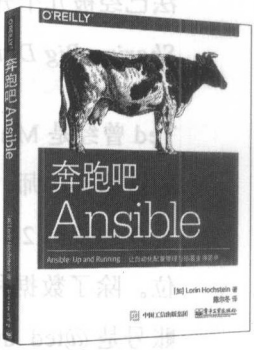
Ellen Friedman，解决方案咨询师，著名演讲者和作家，目前主要撰写大数据方面的著作。她是 Apache Drill 和 Apache Mahout 项目的贡献者。拥有生物化学博士学位，具有多年科研经历，已经撰写过很多技术领域的著作，包括分子生物学、非传统遗传和海洋学。Ellen 还是魔法主题漫画书《戴帽子的兔子》（*A Rabbit Under the Hat*）的共同作者。Ellen 的推特账号是 @Ellen_Friedman。

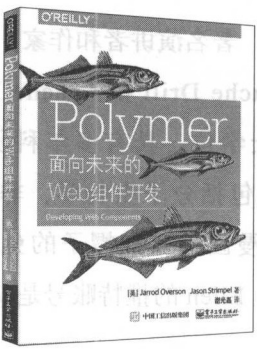
O'Reilly 精品推荐

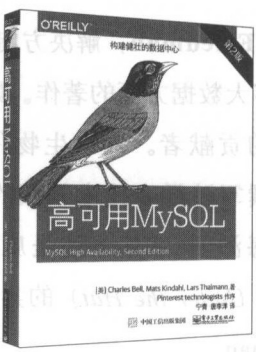




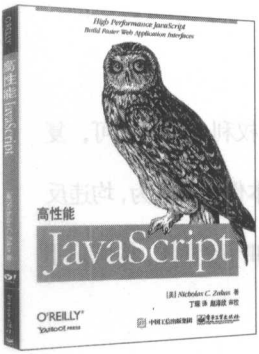


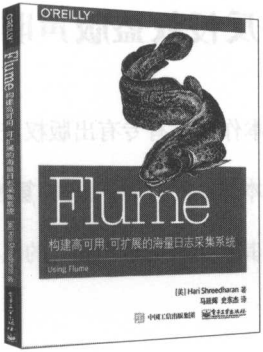


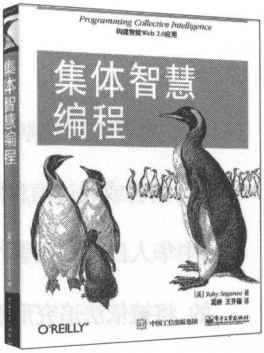


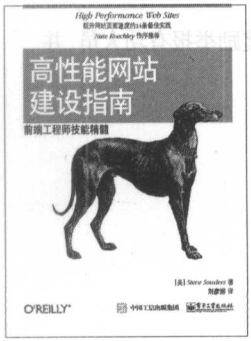


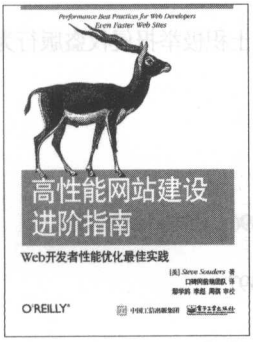
O'Reilly 精品推荐

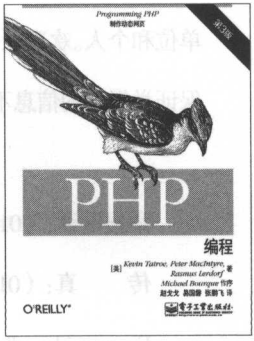


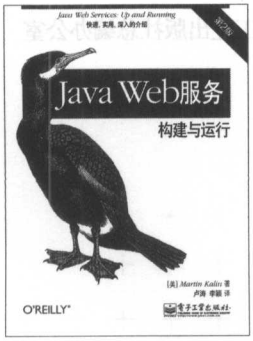


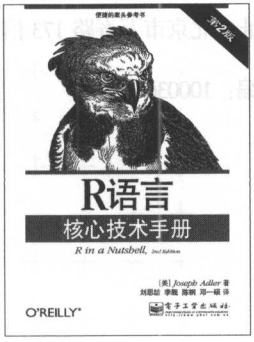












欢迎投稿：

翻译和投稿征集邮箱：
zhangcy@phei.com.cn

编辑微博互动：
http://weibo.com/208686914

更多信息请关注：

博文视点官方网站：
http://www.broadview.com.cn

博文视点官方微博：
http://t.sina.com.cn/broadviewbj

O'REILLY®

流式架构 Kafka与MapR Streams数据流处理

对于数据驱动型公司，设计和构建流式数据架构能够实现实时或近实时应用，提升整个组织的效率。这本简明的指南讲述了流设计中的关键因素（聚焦于消息层的关键特性）、新的消息技术Apache Kafka 和 MapR Streams、流架构是如何支持微服务的，以及当下可供选择的流技术：Apache Spark Streaming、Apache Flink、Apache Storm和Apache Apex，适合架构师、大数据科学家及IT工程师阅读。

本书内容包括：

- 确定使用数据流的最佳场景与时机
- 在多用户系统中如何更好地设计流架构
- 为什么这种设计要求消息传递层具备某些特定的功能
- 为什么流式架构支持微服务
- 最符合流设计需求的消息传递和流分析工具的描述

Ted Dunning，MapR Technologies 首席应用架构师，开源社区的活跃成员。现任Apache Foundation 孵化器的VP。Ted的推特账号是@ted_dunning。

Ellen Friedman，解决方案咨询师，著名演讲者和作家，目前主要撰写大数据方面的著作。她是Apache Drill 和Apache Mahout项目的贡献者。Ellen的推特账号是@Ellen_Friedman。

图书分类：大数据/数据处理

策划编辑：张春雨

责任编辑：徐津平

Broadview®
www.broadview.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-31722-4

9 787121 317224 >

定价：55.00元